TRABALHO 2 DE LINGUAGENS DE PROGRAMAÇÃO

–INTRODUÇÃO:

O segundo trabalho de Linguagens de Programação consiste em programar em C++ uma calculadora de linha de comando que atenda a gramática passada. As operações suportadas são as quatro básicas -divisão, multiplicação, adição e subtração- além de parênteses para precedência e atribuições a variáveis.

Um dos objetivos do trabalho era desenvolver o programa de forma colaborativa, através do uso da ferramenta GIT e do GITHub. Toda a lógica dos módulos e a comunicação entre eles foi, de forma geral, discutida em conjunto e implementada de forma separada. Os diferentes estágios-chave do desenvolvimento da calculadora deveriam ser salvos em um repositório local, e enviados para o repositório global situado no site do GITHub. Porém, por haver uma certa dificuldade em se familiarizar com a ferramenta GIT e com toda a configuração dela e da comunicação com o GITHub em cada máquina de cada integrante, o grupo priorizou o desenvolvimento da calculadora em si, fazendo a troca dos códigos por outros meios, e implementando o código, muitas vezes, de forma presencial, contando com a opinião e ajuda de todos.

–MÓDULOS:

Como já explicado, a lógica de cada módulo do programa foi desenvolvida em conjunto, e implementada em separado, e quando o programador se deparava com alguma dificuldade, a busca pela solução era geralmente feita em conjunto, para que essa solução satisfizesse todos os outros módulos e as comunicações entre si, seja essa uma solução lógica ou específica da linguagem.

Os responsáveis principais por cada módulo são:

–Módulo de validação (validation.cpp):
    Natália França;
–Módulo de manipulação de entradas e strings (dataManipulation.cpp):
    Gabriel Bastos e Luiz Felipe Santos;
–Módulo de operações (op.cpp):
    Luiz Henrique Vasconcelos e Phellipe Moura;
–Módulo de gerenciamento de memória (memoryHandling.cpp):
    Pedro Goñi Coelho;

–FUNCIONAMENTO DO PROGRAMA E DE SEUS MÓDULOS:

A parte principal do programa -main- chama a função Mem::open() para a criação da variável de referência head, que é apenas uma referência como a primeira da lista de variáveis. Depois, chama as rotinas que imprimem na tela o prompt ">>" e depois recebem a entrada do usuário -DataM::getUserInterface-, deletam seus espaços -DataM::delSpaces- e inserem sinais de multiplicação onde eles são implícitos -DataM::insMult, ex: imediatamente antes de parênteses- e então chama uma função de validação da entrada

muito simples, a Val::inputVal, que apenas observa se a entrada foi "quit", utilizado para fechar a calculadora, ou se essa string entrada termina em ";". Caso a validação seja positiva, essa entrada é passada para a função principal de sequenciamento de operações, a Op::run, que opera completamente a entrada do usuário e apresenta o resultado da última sentença. Esses passos são seguidos até que a entrada seja quit, ou a Op::run identifique que uma das subsenteças da entrada é "quit", o que aborta o loop na main, chama a função Mem::close() e termina o programa. A Mem::close() chama uma função que deleta todas as variáveis na memória, mas apenas a própria close deleta o head. Com isso, é possível implementar, em versões futuras, que com uma entrada do usuário a qualquer momento, por exemplo "clear", todas as variáveis na memória sejam deletadas, voltando ao estado inicial "limpo" do programa.

Analisando a função Op::run, vê-se que ela que decide como as operações são feitas. Primeiro ela separa a primeira substring a ser operada -função DataM::subinputVal()- valida essa substring -Val::subInputVal- e a opera se positivamente validada. No fim, a Op::run apaga a substring da entrada do usuário -DataM::rmSubInput- e repete todo o procedimento até que a entrada do usuário esteja vazia, significando que todas as subentrandas já foram operadas. Só então o último resultado é apresentado ao usuário, de acordo com o especificado. A validação também identifica o tipo de sentença, se é apenas uma operação ou se há atribuições. Dependendo de seu retorno, a Op::run executa duas diferentes subrotinas: uma apenas chama a função Op::recursiveHell, que opera normalmente, e a outra separa todas as operações da parte que contém o nome da variável e o sinal de atribuição "=", chamando a Op::recursiveHell. Se o resultado dessa função não for um erro, é criada na memória então uma nova variável que recebe esse resultado.

A função recursiveHell é uma função recursiva, que sempre separa as operações de uma em uma. Ela é o "cérebro" do programa. Ela primeiro observa precedências, buscando parênteses, divisões, multiplicações, adições e subtrações, nessa ordem. Assim, ela elimina todo o resto da entrada, opera no nível mais baixo, simples, possível, e substitui na entrada original o resultado, repetindo o processo até que não hajam mais operações a serem realizadas.

Quando o nível mais baixo é atingido, é chamada a função Op::operate, que opera de fato. Ela busca a posição da operação desejada, observa se algum dos operandos é uma variável -Mem::search-, substituindo seu nome pelo valor na memória, identifica novamente a posição da operação, que pode ter mudado com a substituição feita, converte ambos os operandos de string para double -DataM::stringToDouble- e chama a função operation.

Uma decisão importante tomada foi substituir os sinais de número negativo de "-" para "#", para não confundir os números negativos com as operações de subtração. Assim, sempre que o número precisa ser convertido para double ou mostrado para o usuário, antes é feita a troca de caracter e após a aplicação a substituição é refeita.

A função operation recebe o char que é o sinal de operação, e ambos os operandos em double. Essa função é apenas um conjunto de testes "if" que identificam a operação e retornam o resultado. Cabe ressaltar que se a operação for divisão e o divisor for zero, uma operação indefinida, foi decidido retornar o valor zero. Isso foi feito para se evitar ter que tratar o resultado "inf" -infinite- que a biblioteca padrão "math.h" retorna nesse casos. O usuário, porém, é avisado através de uma mensagem na tela que o resultado está errado por ter caído em uma divisão por zero, indefinida, e que deve ser desconsiderado.

O módulo de validação, além do já explicado, faz uma busca por vários erros de sintaxe em cada substring, na função Val::subInputVal. Essa função busca uma entrada

"quit"ou uma atribuição. Se encontrada uma atribuição, ela separa a parte da atribuição e e retorna uma função mais geral, sentenceVal, para validar. Caso não haja atribuição, é retornado o resultado da sentenceVal para o input inteiro. A sentenceVal em si faz um número de buscas de erros de sintaxe na subentrada, como número inconsistente de parênteses, ou se eles estão apropriadamente fechados, posições inválidas para operações -sinais de multiplicação seguindo abertura de parênteses- ou operações seguidas, etc.

Finalmente, sobre o módulo de manipulação da memória, que gerencia a lista de variáveis criadas, vale acrescentar o funcionamento das funções Mem::add e Mem::search além de especificar o tipo de variável guardado na memória. O tipo utilizado é um struct chamado de "my_var". Esse struct contém 3 campos: uma string "nm", que guarda o nome da variável; uma string "val", que guarda seu valor; e uma referência para a my_var anterior, my_var *prev, o que implementa uma lista encadeada de variáveis. Todo o módulo de memória trabalha retornando apenas valores, assim, os módulos de níveis superiores não precisam fazer a manipulação de ponteiros.

A add adiciona sempre uma variável no fim da lista, mantendo sempre atualizadas a posição do "last", uma referência a última variável da lista.

Já a search sempre procura a variável desejada de "trás para frente", ou seja, fazendo uma variável referência ao tipo my_var chamada "current", percorrer a lista desde a posição "last" até o "head", sempre pulando para a referência "prev" do current quando não encontrado. Se o current encontrar o nome desejado, retorna-se o seu campo val. Caso contrário, se o current alcançar o head, retorna-se um erro.

–CONCLUSÕES:

Como avaliação final do programa, o grupo acredita que ele seja completamente funcional, salvo de alguns casos não previstos, apesar de terem sido feitos diversos testes. Acredita-se também que o módulo que faz as operações de fato esteja completo, e que eventuais problemas venham de alguma entrada não prevista pelo módulo de validação de entrada. Além disso, quando é feita uma atribuição, o programa sempre cria uma nova variável na memória, ao invés de procurá-la na memória e sobrescrevê-la caso já exista. Isso implica em um uso excessivo de memória, visto que o mesmo espaço poderia ser reaproveitado. Como o tipo de variável utilizado não ocupa um espaço tão significativo comparado com a memória dos computadores atuais e como a busca é sempre feita desde a última variável incluída, ou seja, a variável achada é sempre a última a ter sido incluída, não comprometendo o uso da calculadora, foi decidido não modificar essa parte para não atrasar a data de entrega do trabalho. Porém o problema foi identificado e a solução facilmente encontrada, podendo ser implementada em futuras versões do programa. Além disso, o módulo de validação prevê variáveis com nomes de apenas um caracter, para facilitar o trabalho do módulo de memória. Porém, o módulo de memória, por ter sido feito por último, já aplica soluções que contornam os problemas previstos. Por isso, acredita-se que para o uso de variáveis com nomes maiores, pode-se apenas fazer modificações no módulo de validação e talvez outras pequenas modificações em outros módulos. Logo, em futuras versões isso pode também ser facilmente implementado.