四川大学计算机学院 （软件学院、 智能科学与技术学院）

实验报告

学号：x 姓名： x 专业： 人工智能 班级： 233040701 第 18周

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程名称 | 数据结构与算法分析课程设计 | | 实验课时 | 4课时 |
| 实验题目 | 课程设计（simple-matlib） | | 实验时间 | 2024/12/31 |
| 实验目的和要求 | 本课程设计实现了针对数学函数的基本的符号/数值计算。并设计了一个简单的语言，并实现了一个简单的解释器来理解该语言，完成相应的数学计算。  **数学函数支持如下：**   1. 数（整数、小数） 2. 变量：用小写的ascii字符串表示，如x、length、weight 3. 函数：用大写的ascii字符串表示，支持SIN、COS、LN、EXP 4. 操作符：四则运算+-\*/，以及!(阶乘),^(幂运算)   **解释器功能如下：**  Note：在下文中注意区分**变量**与**数学符号**，下文中**变量**代表着一个数学函数（数学表达式），而**数学符号**是存在于数学函数中的。例如，**let a= 1+2+x**，a是变量，类型是数学函数，代表这个函数**1+2+x**，而其中的x是数学符号。   1. 注释   **[code] #your comment**   1. 为变量赋值数学函数，使用**let语句进行特殊赋值**   **let [variable]= [math-function] let a= 1+2+x #将数学函数1+2+x赋值到变量a上**   1. 变量拷贝   **[variable]= [variable] #数学函数复制 b=a**   1. 求导/求偏导（符号计算）   **deriv [variable] [math-symbol] deriv b x #对变量b代表的数学函数求关于数学符号x的导 c=deriv b x #对变量b代表的数学函数求关于数学符号x的导，并将结果赋值给c**   1. 求值（数值计算）   **eval [variable] x:1 y:2  eval b x:1 y:2 #对变量b代表的数学函数求值，其中数学符号x=1,y=2 d=eval b x:1 y:2**   1. 化简（符号计算）   **sim [variable]  e=sim b #化简b并将结果存入变量e中**   1. 一维定积分（数值计算）   **int [variable] [math-symbol] [from] [to] [precision] int b x 0 1 0.01 #将变量b代表的数学函数在区间0~1间按照步长0.01积分 f=int b x 0 1 0.01 #...并将结果存入f**   1. 退出   使用exit命令即可退出整个程序。  除了上述基本功能之外，本程序还实现了鲁棒的异常处理，尽可能在出现异常的地方进行检测并包装为Exception抛出。整个主逻辑都在try-catch块中。  为了更好地检视每一步操作的详情，还可以在启动时传入debug参数，此时程序会以debug模式启动，会在控制台中输出更详尽的内容。  由于在程序编写过程中垃圾回收策略存在一定的小bug（当然现在都修复了），因此本程序依旧保留了对nogc参数的支持，在此模式下，程序不会启用垃圾回收。 | | | |
| 实验环境 | Clion2024、MinGW11.0、C++17 | | | |
| 算法描述 | 1. 整体概览 2. 封装解释器类用于处理用户输入，并生成输出。这个类的主要功能就是基本的动态语言解释器，其功能如下： 3. 指令解析：将用户输入的字符串指令解析成程序可以理解的内部表示。 4. 变量表管理：创建和维护一个变量表，用于存储和管理变量及其对应的数学表达式。   其中，对于变量表管理，可以使用映射（Map）实现，实现变量名（string）到数学函数(AstNode\*)的映射关系。  并且为了不造成内存泄漏，我还为其中的变量实现了一种垃圾回收机制，在变量即将不可达时，及时释放其占用的内存区域。   1. 异常处理：检测并处理在执行过程中可能出现的异常情况。 2. 调试模式支持：在启动时接受debug参数，以输出更详尽的调试信息。 3. 无GC模式支持：在启动时接受nogc参数，以禁用垃圾回收。   其简要流程图如下：     1. 为了解析数学函数，这时候就要用到堆栈了。先将字符串进行预处理得到一系列分割的字符串（也就是中缀表达式），再转化为后缀表达式。然后再通过后缀表达式构建树。其算法原理我在《实验一报告》中有详细说明，在此仅简要说明： 2. 首先，将输入的数学表达式字符串按照操作符、空格等分隔符进行分割，从而得到一个由各个数学元素（数字、变量、操作符等）组成的字符串列表。这个列表代表了中缀表达式的结构。 3. 接着，利用两个堆栈——一个用于存储操作符（操作符栈），另一个用于存储操作数（操作数栈）——将中缀表达式转换为后缀表达式。这个过程遵循特定的算法规则，确保操作符的优先级和结合性得到正确处理。最终，后缀表达式以字符串列表的形式表示，其中每个元素的顺序反映了计算的顺序。 4. 最后，使用一个堆栈来构建二叉树。从后缀表达式列表中逐个取出元素，根据其类型（数字、变量、操作符等）创建相应的AstNode节点。对于操作符节点，需要从堆栈中弹出两个元素作为其子节点，然后将该操作符节点入栈。这个过程持续进行，直到后缀表达式列表被完全处理，堆栈中最终只剩下一个节点——即整个表达式的根节点。将这个根节点取出，就完成了对应数学函数的二叉树构建。   为了方便二叉树的各种操作，简化函数实现，我将各种类型的结点都统一为AstNode。但其在内部分为如下4种类型：   1. 操作符类型：代表数学中的加、减、乘、除等操作符。这些节点包含左右两个子节点，分别代表操作数。 2. 函数类型：代表如SIN、COS等数学函数。这些节点也包含子节点，代表函数的参数。 3. 数学符号类型：代表变量，如x、y等。这些节点是叶子节点，没有子节点。 4. 立即数类型：代表具体的数值，如整数或小数。这些节点也是叶子节点，没有子节点。   树的节点结构分类如下图所示  而树的一个结构实例如下：    可以看到，操作符、函数作为内部节点串起所有的立即数、数学符号。通过这种方式，我们可以将复杂的数学表达式分解成易于管理和计算的结构，从而实现对表达式的求值、求导、化简和积分等操作。   1. 数学函数求值：这时候就需要通过后序遍历了，自下而上地求值。最终计算出整个表达式的值。在程序设计上，这可以通过递归函数实现，只需要稍加区分节点的具体类型即可： 2. 操作符节点：对于操作符节点，我们需要根据操作符的类型（如加、减、乘、除）和子节点的值来计算当前节点的值。 3. 函数节点：对于函数节点，我们需要根据函数的类型（如SIN、COS）和子节点的值来计算当前节点的值。 4. 数学符号节点：对于数学符号的，需要通过查表得到，这个数学符号表也是一个映射（Map<string,double>），在求值时，需要传入此表。 5. 立即数节点：它们的值是确定的，无需进一步计算。 6. 数学函数求导：这个操作乍一看感觉很复杂，但稍加分析就会发现这个其实不难。数学函数求导具有很清晰明了的公式，就拿乘法操作为例，若y=a\*b,那么y’=a\*b’+a’\*b，而a’、b’又可以通过这个操作递归地算出。这在本质上就是数学上求导的链式法则。   实际上，对于内部节点（函数操作、算数操作）都是如此，不必关心子节点到底是哪一种类型，将它们都视作抽象结点即可。只需要对叶子节点特别关注即可：立即数的导数始终为0、数学符号对需要求导的数学符号导为1，其他则为0（例如，对于数学符号x、y有：）。  该算法的图示如下（以乘法操作为例）：  如果使用遍历树的视角，这在某种程度上可以看作是自顶而下的，先读出根节点的值，然后根据节点的类型（如加法、乘法、指数等）应用相应的求导规则来计算当前节点的导数，从上向下地构建导数的结构。  通过这种方式，求偏导也变得很容易。因此，在多变量函数中，求导可以针对特定的变量进行，这在物理问题中尤为重要。   1. 数学函数化简：在数学上，我们知道0是实数域上的乘法零元，有0\*a=0；而1是乘法幺元，有1\*a=a；对于加法，0是幺元，有0+a=a；如果结点都是立即数，那么可以先将结果算出。除了上述提到的几种操作，对于除法、减法也有类似的特殊情况。数学函数的化简得以实现。   对于0+a=a的图示过程如下:  对于子节点都是立即数的化简图示过程如下：    为了将整棵树进行化简，可以通过后序遍历的方式将这种操作自下而上地应用于每个结点。虽然这个操作会破坏树的结构，但其实这并不会影响树的遍历，因为遍历顺序的下一个值不会依赖于已经更改的节点。   1. 积分运算：由于符号积分在数学上可能非常复杂，尤其是在处理高阶多项式、三角函数或其他特殊函数时，本程序选择使用数值积分方法，以简化计算过程。   按照积分的定义，可以设置一个很小的步长。步长决定了积分的精度，较小的步长可以提高积分的精确度，但同时也会增加计算量。通过将积分区间划分为多个小区间，并在每个小区间的某个点上计算函数值，然后使用这些值来近似整个区间的函数面积。然后通过累加所有小区间的函数值，我们可以得到整个积分区间的定积分近似值。   1. 关键具体算法实现 2. 函数化简   以下函数实现了对单个节点进行化简的操作。  **static void *simplifySingle*(AstNode\*\* node\_raw){  AstNode\* node=\*node\_raw;  if(node->op=="+") {  if(node->left!=nullptr&&  node->left->type==immediate&&  node->left->immediateNumber==0){  *destroySingle*(node->left);  *destroySingle*(node);  \*node\_raw=node->right;  return;  }**  **……**  记当前节点为A，对于上述情况，A的左节点为0，需要将A的右节点提到当前节点的位置。  但由于这个二叉树不是线索二叉树，无法从A获取其父节点，似乎也就无法为A的父节点设置其子节点为A的右节点。  但其实不然，这可以通过使用C语言中强大的内存操纵者——指针实现。在这里，这可以通过**二维指针**进行操作，我们传入&node->left（或right）即可，利用指针的特性更改父节点的子节点  **static void *simplify*(AstNode\*\* root) {  const auto node=\*root;  if (node==nullptr) {  return;  }  if (node->left!=nullptr) {  *simplify*(&node->left);  }  if (node->right!=nullptr) {  *simplify*(&node->right);  }  AstNode::*simplifySingle*(root); }**  由于simplifySingle函数只能化简子节点都是叶子节点的节点，因此我们还需要实现一种遍历操作将这个化简操作作用于每一个节点，在此我们使用一个递归函数来实现。值得注意的是这个遍历的顺序是有讲究的，需要自下而上地进行化简。   1. 求导/偏导操作   **AstNode\* derivative(const string& variable) {  AstNode\* result = nullptr;  if (type == ElementType::immediate) {  result= new AstNode(ElementType::immediate);  result->immediateNumber=0;  }  else if (type == ElementType::variable) {  result = new AstNode(ElementType::immediate);  if (variableName == variable) {  result->immediateNumber = 1;  } else {  result->immediateNumber = 0;  }  }  else if (type == ElementType::function) {  if (op == "SIN") {  auto cos=new AstNode(ElementType::function);  cos->op="COS";  cos->left=left->copy();  result= *mul*(cos,left->derivative(variable));  }**  **……**  **}  else if (type == ElementType::op) {**  **……  }  return result; }**  对于求导操作，我们同样也可以使用递归函数实现。只需要传入一个参数，也就是要求偏导的数学符号（如果只有一个数学符号那么就是求导）。可以看到，只需要在**derivative**函数中为当前节点的具体类型进行分类操作，而不必考虑子节点的具体类型是什么。 | | | |
| 源程序清单 | main.cpp 程序入口，解析命令行参数  Exception.h 定义基本异常类型，便于统一处理异常  Interface.h 实现处理用户输入输出解析的封装，也就是解释器模块。  ExpressionHandler.h 用于管理运算表达式，实现表达式解析、toString等操作  AstNode.h 实现基本的运算表达式结点，并实现求导、求值等算法。  Utils.h 工具类，实现字符串分割，从控制台读取输入等操作 | | | |
| 运行结果 | **基本使用:**    可以看到，在倒数第二条指令，即便我不小心将格式打错了，程序也立即进行了错误提示。  **Debug模式展示（在启动参数中加入debug）**：    使用debug模式，程序会输出详细信息。可以看到解释器对指令的预处理。在执行  b=sim b指令时，还可以看到对之前b对象的垃圾回收。  **极端情况尝试：**  错误表达式尝试：    错误指令尝试：    复杂导数尝试：    即便在极端情况下，程序的运行也非常可靠。能准确地给出错误提示。 | | | |
| 实验运行情况分析(包括算法、运行结果、运行环境等问题的讨论) | 程序运行符合预期。  **内存管理上存在不足：**  本程序在内存管理上存在不足。由于内存释放是由解释器管理的，解释器会直接将不可达的变量的内存全部释放，也就是说，这个数学函数会链式地从根节点开始释放掉所有的节点，内存释放的基本单位就是整棵树，而不是节点。这乍一看好像没有问题，但实际上这其实是很简单粗暴的，因为很多时候**内部节点理论上是可以复用**的。  就拿求导举例，假设变量a代表数学函数f\*g（f、g表示抽象节点，代表子树）。显然a的导函数a’=f\*g’+f’\*g，子节点f、g理论上是可以复用的，但实际上我们并不能直接将其复用，因为其可能会被释放，如下所示：  那么我进行如下操作：  a = deriv a x # 将a代表的数学函数对x求导，并将结果写入变量a  此时，之前的那个a变得来不可达了，解释器会将之前的a代表的数学函数整个释放，也就是说其子树f、g也被释放了。那么，在这样的垃圾回收模式下，并不能保证子树的完整性，也就不能复用子树。这样就造成了大量的对象复制，消耗了额外的CPU时间与内存空间。  改进方法肯定是有的，我们可以将垃圾回收的单位进一步缩小、精细化。可以将垃圾回收的单位缩小到单个节点，然后实现引用计数，就像java、python等语言一样，但这样的工作量又太大了，对于这样一个小程序来说有点本末倒置了。  **一个Bug的修复：**  在程序测试过程中，发现有时候会出现段错误（SIGSEGV），经过调试，发现该错误出现在内存释放代码中。这部分刚开始的逻辑是链式释放，代码如下：  **static void *destroy*(const AstNode\* root){  if(root->left!= nullptr){  *destroy*(root->left);  }  if(root->right!= nullptr){  *destroy*(root->right);  }  delete root; }**  经过仔细思考，我虽然不能也没有直接复用之前的二叉树的子树节点，但为了提高效率，在一棵二叉树的内部，是复用了节点的，并不是在每个地方都对子树进行了拷贝。例如，在除法的求导过程中，在理想的代码逻辑中，二叉树的形状应当如下：    但在实际实现中，为了提高内存利用率，红色部分的抽象节点B实际上是同一份拷贝。也就是说，这个树的形状是这样的：    虽然在我们的算术逻辑上，这应当是一棵树，但实际上由于节点复用，导致了这在存储逻辑中不是一棵树。因此在这种情况下，该函数就会访问已经释放过了的节点，那么就会访问其中的野指针**root->left**，造成段错误与程序崩溃。理清了这个思路，那么该如何修复这个bug呢。  我的想法是在该对象中放一个ValidTag（32bit）来标明有效性，并认定如果其值为0x12345678这个对象就是有效的。然后在销毁对象时破坏这个ValidTag的有效性。即便这块内存区域塞入了别的数据，这个ValidTag也很难与之前的完全一样（若填入的数据随机，概率约为2.3283e-10，这可以认为是不可能的）  在AstNode类中添加如下代码  **int validTag=0x12345678;**  **~ AstNode() {  validTag=~validTag; }**  然后在destroy函数中添加有效性检验  **static void *destroy*(const AstNode\* root){  if (root==nullptr||root->validTag!=0x12345678) {  return;  }  if(root->left!= nullptr){  *destroy*(root->left);  }  if(root->right!= nullptr){  *destroy*(root->right);  }  delete root; }**  这个bug在本质上还是因为内存控制不够精细造成的。这个bug听起来很简单，但实际上我真的找了好久，解决方案也尝试了很多次。 | | | |
| 指导老师  评议 |  | | | |
| 成绩评定： | 指导教师签名： | | |