**编译原理 实验报告**

|  |  |  |
| --- | --- | --- |
| **评 语（4号楷体）** | **成绩** |  |
| 教 师：  年 月 日 | | |

**教学班级：**

**学生学号：**

**学生姓名：**

**实验日期： 2024.12**

**一、实验目的**

通过做上机题加深对编译器构造原理和方法的理解，巩固所学知识。

(1)会用正规式设计简单语言的词法；

(2)会用产生式设计简单语言的语法；

(3)会用递归下降子程序编写语言的解释器；

(4)使用递归下降算法编写解释器,加深理解

(5)更加熟悉通过面向对象方法编写程序

**二、实验环境（软/硬件环境）**

软件环境:

操作系统:Windows 11 家庭版

编译器:mingw64

IDE: VSCode

硬件环境:

处理器 12th Gen Intel(R) Core(TM) i5-1240P

已安装的物理内存(RAM) 16.0 GB

**三、实验内容（设计/实现思路，即如何完成的。不允许贴大段代码）**

**3.1设计思路(总)**

整个项目总共有两个模块,词法分析器scanner和语法分析器parser,然后再加上一个主程序main.其中语法分析器起主导作用,由语法分析器来调用各个模块,词法分析器提供Token流,主程序main绘制窗口,然后叫语法分析器上班.

语法分析器分析绘图语句,当分析完语句FOR … TO … STEP … DRAW,就按照获得的参数绘制图形.

在编写程序之前,我先想好接口该怎么设计,也就是先写好两个.h文件,scanner.h和parser.h .然后再实现scanner.cpp,再在scanner文件夹下编写测试程序test.cpp,测试词法分析器是否工作正常,最后实现parser.cpp,然后进行整个程序的测试.

这个程序涉及了多个源文件编译,所以在编译的时候采用Makefile提前写好编译方式,将每个模块分别编译为.o文件后将他们链接起来,这样的好处是编译时需要的命令简洁,还有编译出现错误时,可以更容易排查.

考虑到解释器运行时会频繁申请和释放内存,而在C++中使用new和delete会有导致内存泄漏的潜在风险,这个解释器使用的是C++的新特性---智能指针,它能够自动管理对象的生命周期,在对象失去所有引用后释放掉这个对象的内存(后面这个智能指针搞出了十分离谱的错误,花费了我大量时间排查).

为方便起见,这个解释器绘图调用windows自带的绘图接口GDI(Windows的祖传代码)

**3.2词法分析器**

在词法分析器里声明了Token类型,Token\_type类型,符号表TokenTable作为全局变量,主要内容是定义了两个类,一个是scanner类,一个是DFA类.

首先说scanner类,它的构造函数接受一个文件名,然后利用这个文件名创建一个ifstream对象读取目标文件,在析构函数关闭这个ifstream.

它的成员函数getToken()实现了这个分析器的主要功能,它通过DFA状态转移实现分析,getToken()保存两个DFA状态,一个是之前的pre\_state,一个是读取一个字符进行状态转移后的state,如果move()函数发现这是个错误的字符输入,那么getToken()就会检查之前的pre\_state是不是终态,如果是,就当现在这个字符没有读到,放回ifstream里,然后返回当前读取到的字符串对应的Token.

然后是DFA类,他有两个成员变量,一个是map<pair<int,char>,int> dfa,

它储存dfa的状态转换,由于是哈希表,所以在确定状态转移时可以直接得到下一状态.另一个成员变量是Token\_Type final\_states[14],数组下标的值就是状态的值(状态是用int表示),通过这个数组可以直接获得终态类型.

dfa类有两个成员函数,一个返回状态转移int move(int src,char tag),

一个返回终态类型Token\_Type finalState(int src,string buffer,Token& token),返回终态类型的接受了一个Token类型的引用,可以在查符号表时顺便给Token赋上可以确定的值.

**3.3语法分析器**

在语法分析器中包含两个类加上一个绘图接口函数,一个类是ExprNode,这个类表示一个语法树的节点,另一个类是parser,语法分析器的主体,他从scanner中取得Token流,并对这个Token流进行语法分析,构建语法树,获得绘图必要的参数,并且计算绘图位置,调用绘图接口函数.

先说ExprNode类,这个类储存了一个语法树节点的类型,与这个节点可能存在的左右子树或者值,它拥有两个成员函数,一个是getValue()函数,通过对自己作为根的树的中序遍历返回这个树的值,另一个成员函数是printTree()函数,他是用于调试的,负责打印自己作为根的树的中序遍历,这两个函数都是递归的

ExprNode类有四个构造函数,他们的参数表各不相同,不同构造函数对应不同的节点类型.

然后是parser类,它是语法分析器,语法分析的逻辑大体上是:判断语句类型(ORIGIN ROT SCALE FOR),匹配相应的关键字,剩下的都是表达式.然后是另一部分:表达式的计算,通过Expression(),Term(),Factor(),Component(),Atom()层层向下,区分计算的优先级.他们的逻辑是递归的,比如Expression()只管加减,在它眼中,表达式就是:xxx {+/- xxx},这个xxx是什么他不管,反正是一个Term,调用Term(),叫他返回xxx是什么样的Term(就是返回xxx的语法树),然后按照xxx {+/- xxx}的逻辑,把Term()返回的树当作子树,组成新的语法树返回,其他的函数做法逻辑上都一样,只是在他们眼中的文法不一样,比如Term眼里是xxx {\*或/ xxx},直到Atom,返回一个最基本的单元,但如果Atom发现一个左括号,他知道又来一个表达式,于是又会调用Expression(),如此这般,直到一条语句分析完毕,然后接着下一条.

当语句来到FOR … TO … STEP … DRAW时,parser就会调用drawLoop(shared\_ptr<ExprNode>left,shared\_ptr<ExprNode>right)来绘制图形,在绘制一个点之前,调用cal\_dot(shared\_ptr<ExprNode> left,shared\_ptr<ExprNode> right)来计算这个点旋转,拉伸,平移之后的实际坐标.

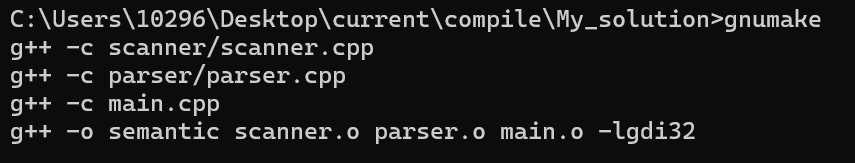
Parser类还带有简单的错误处理函数,指出错误出在哪一个地方.

**3.4主程序**

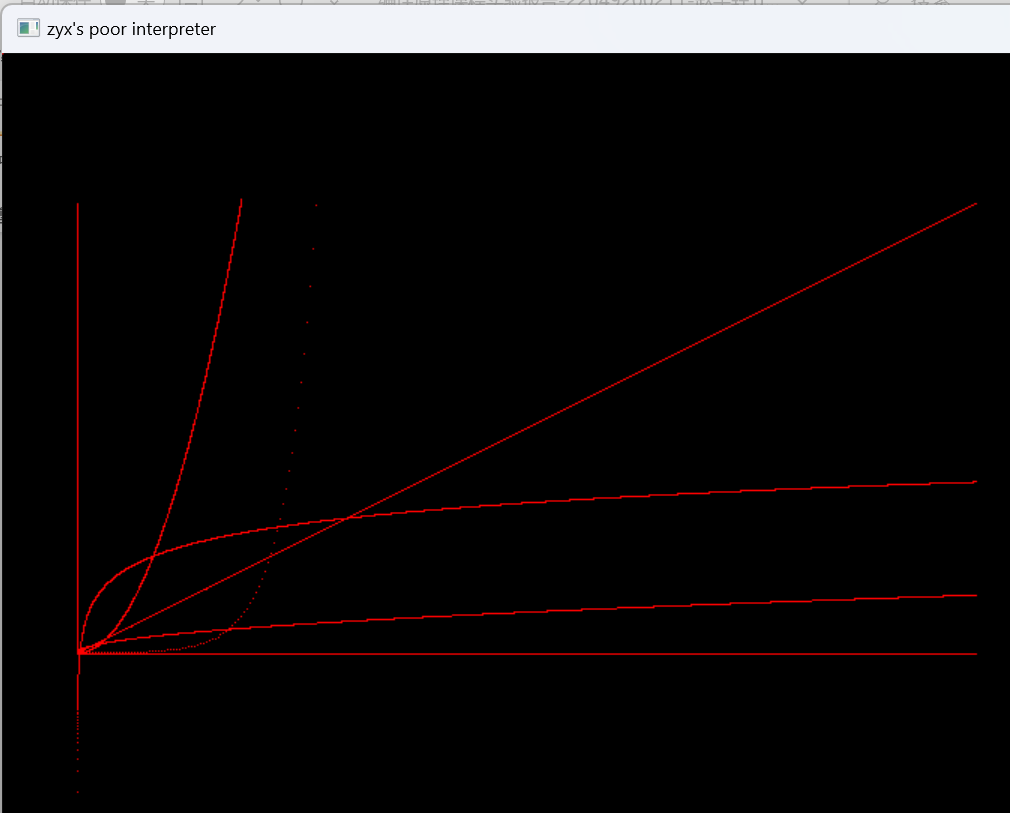
就是一个win32程序的简单框架里头调用了一下parser

**3.5项目编译执行**

贴两张图验证一下

解释器编译:

执行:



**四、心得体会（完成时遇到哪些问题以及问题是怎样解决的、目前的做法的优点及不足）**

**4.1问题**

**问题1**

遇到的问题主要有两个,先说第一个,就是之前提到的智能指针,刚刚写完,编译好,自信满满开始测试,结果程序总是画一半就崩溃,而且每次崩溃前画的进度都不一样,有时完整画完了正常结束(画的都是同一份文件),通过让程序输出中间结果,发现数据存在丢失:之前好好的分析出一个值,分析完打印语法树一看,变成0了,对代码上下其手,检查一遍又一遍,看不出问题.最后猜测是这个语法树节点的存储处理问题,如果是这个问题,只能是union的错(union存节点信息),改成struct,欸,好了.推测当我通过构造函数赋值union的shared\_ptr<ExprNode>类型会导致shared\_ptr 机制识别不到union的shared\_ptr<ExprNode>,因为他不知道这个union里头到底是存放了什么类型,也有可能是赋值进union不会增加shared\_ptr 的引用数量,于是,这片空间就这样被释放掉了....之前的分析直接白搭.(问了GPT,关键在于需要显式调用union的析构函数,智能指针的引用才会减少,直接修改union某个成员不会导致析构函数的调用)

**问题2**

编译时总是有报错:多重定义,搞不明白,问问GPT,回答是头文件有定义,每个cpp文件include一下就定义一次,我想,不是有#ifndef和#pragma once保护吗,结果实在搞不定,于是把声明和定义严格分开,.h文件就声明,终于编译通过了.书上说的真的有用啊…

**4.2优点与不足**

优点是面向对象方法,结构清晰,可拓展性好

不足是ExprNode类里头用的是struct而不是union,多占了一点空间