## 主要数据结构设计及实现

本此题目需要处理的对象主要有词法元素、语法元素（包括语句和表达式）、作用域以及最终结果，对应的基类分别为Token、Statement、Expression、Scope和Result. 以下分别对这些类及其派生类的设计思想和实现作简要介绍。

### Token类（lexer.h: Class Token）

Token类是本次题目中会出现的所有词法元素的抽象，其主要属性有：

linum: 所在行号

code: Token的具体内容

type: Token的类型

该类在程序中作为词法分析阶段的结果以及语法分析阶段的输入而存在。

### Statement类及其派生类(parser.h: Class Statement)

Statement基类描述的是C语言中“语句”的概念，它有一个基本动作：

virtual void execute(Result& result, Scope& scope)

该函数对自身所代表的语句在scope代表作用域中进行执行，并将结果存放到result中。

该类本身不实现任何功能，具体功能由其派生类进行实现（见execute\_stat.cpp），其派生类有：

CompoundStat：复合语句

DeclStat：声明语句

SelectStat：if..else语句

ForStat：for语句

WhileStat：while语句

BreakStat：break语句

DoStat：do…while语句

PrintStat：printf语句

ExprStat：表达式语句（为方便实现，空语句也归为此类），如”3+4;”、”a+++b++;”等

语句可嵌套，且若某一语句内嵌套了其它语句，嵌套的语句的execute函数会按照题目所规定的控制流被递归地执行，并且会尽量用C++中相应的语句来模拟其控制流（比如用for语句来模拟for语句），又因为C++与C的基本控制流语句并无出入，所以可以保证控制流的流向与C表现一致。以WhileStat的execute函数(execute\_stat.cpp: void WhileStat::execute(Result& result, Scope& scovoid WhileStat::execute(Result& result, Scope& scope) )为例:

void WhileStat::execute(Result& result, Scope& scope)

{

while(result.add\_line(linum), expr->eval(scope)) {

try { stat->execute(result, scope); }

catch (BreakException e) {break; }

}

}

该函数用了while循环来处理while循环，并且在会被执行到的部分以逗号表达式的形式输出了一条行号到结果中，并在循环体内执行其循环体中的语句。值得注意的是，我们使用了异常处理机制来处理break，这种处理方式很自然，而且与break语句的实际表现一致。

对于Scope，我们会在出现新作用域的时候（如CompoundStat）新建一个Scope对象，并将其parent域指向当前Scope，然后将新建的Scope对象的引用传给内层语句继续执行。

### Expression类及其派生类（parser.h: Class Expression）

Expression基类描述的是C语言中表达式的概念。它有一个基本动作:

virtual long long eval(Scope &scope);

该函数对自身所代表的表达式在scope指定的作用域内进行求值，并返回其结果。代码所使用的基本数值类型为long long，以保证能够容纳题目中出现的数值。

其派生类大致可分为双目表达式，单目表达式以及初等表达式（变量名、常量），单双目表达式的操作数仍然为一表达式。派生类如下：

CommaExpr： 逗号表达式

EqualityExpr：相等性表达式（==, !=）

RelationalExpr：关系表达式（>,<,>=,<=）

AdditiveExpr：加法减法表达式

MultExpr：乘法除法表达式

UnaryExpr：正负表达式

PostfixExpr：自增自减表达式（后置）

PrimaryExprConst：变量

PrimaryExprId：常量

派生类的eval函数实现在eval\_expr.cpp中。现以AdditiveExpr为例介绍eval函数的一般实现思路：

long long AdditiveExpr:: eval(Scope& scope)

{

long long val1 = expr1->eval(scope), val2 = expr2->eval(scope);

if ("+" == op) return val1 + val2;

else if("-" == op) return val1 - val2;

else return -1;

}

该函数先递归地对左右表达式进行求值，然后根据操作符返回相应计算结果。

### Scope类（parser.h: class Scope）

该类描述的是C语言中变量作用域的概念。其属性如下：

Scope\* parent：指向其父作用域的指针，若为顶层作用域，则为parent为nullptr

std::unordered\_map<std::string, long long> vars：从变量名到值的映射

动作如下：

long long get\_identifier(const std::string& name);

void add\_identifier(const std::string& name, long long val);

void mod\_identifier(const std::string& name, long long val);

其中get\_identifier会从当前的键值仓库中查找名字为name的变量，若找不到，则递归地从父scope中查找。add\_identifier会直接在当前作用域中添加一新的键值对<name, val>（通常用于变量定义），而mod\_identifier则会递归地从当前作用域和父作用域中找到包含name的作用域并将其值改为val（通常用于赋值语句）。

### Result类（parser.h: class Result）

该类用以存放最终结果。该类有两个动作：

void add\_line(int linum);

void print(std::ofstream& ofs);

其中add\_line被实现为当且仅当上次添加的行号不等于这次添加的行号才将linum放入最终结果中。

## 模块划分及功能实现

程序被划分为四个阶段：输入，词法分析，语法分析及结果计算，输出。其中词法分析被封装在Lexer类中，相关文件为lexer.cpp, lexer.h; 语法分析及计算被封装在Parser类中，相关文件为parser.h, parser.cpp, parse\_stat.cpp, parse\_expr.cpp, stat\_execute.cpp, expr\_eval.cpp。下面对词法语法分析以及结果计算部分的设计和实现分别予以简要描述。

### 词法分析（lexer.h: class Lexer）

该类的核心函数为scan（lexer.cpp: int Lexer::scan(string s, vector<Token>& result)），其功能为将读入的字符串转换为一个Token列表以供后续过程使用。

scan的核心部分为一个循环，其框架如下：

1. 遍历模板列表，从剩余字符串中识别并提取出下一个token
2. 根据读出的token类型及内容，确定是否增加当前行号以及增加多少
3. 根据独处的token的类型，确定是否将当前token加入最终token list中

循环结束后，返回最终Token list。

为了降低人工编写状态机出错的可能性，同时也为了让代码易于扩展，其中的token识别以及提取使用了正则表达式来进行。相关的token类型及其对应的正则表达式可在Lexer类的构造函数（lexer.cpp: Lexer::Lexer()）中找到。

### 语法分析及结果计算（parser.cpp: class Parser）

该类的核心函数为parse（void Parser::parse(Result& result)），该函数对token list进行语法分析并计算，然后将结果填入result中。为了降低人工编写状态机出错的可能性，同时也为了让代码易于扩展，该函数首先用一个Token::ToeknType到char的映射表对token list进行编码，于是使得后续的分析可以使用正则表达式在这个编码得到的字符串中进行。

首先介绍语法分析，语法分析根据编码后的字符串，将token list转换成一个大的CompoundStat对象。

语法分析的核心函数为：

Parser::parse\_stat(string::const\_iterator str\_begin,

string::const\_iterator str\_end,

size\_t origin,

shared\_ptr<Statement>&stat\_ptr);

该函数根据str\_begin指向的编码后字符内容判断出下一个语句的类型，然后将当前待分析字符串分派到相应语句的parser函数，并将parse后得到的Statement对象的指针赋值给stat\_ptr。不同语句的parser函数具有统一的形式，其形式被抽象为一个宏：

#define STAT\_PARSER\_PROTO(T, fn) \

std::string::const\_iterator \

PARSER\_NAME(fn)(std::string::const\_iterator str\_begin, \

std::string::const\_iterator str\_end, \

size\_t origin, \

T& stat)

其中的PARSER\_NAME具有如下形式：

#define PARSER\_NAME(fn) parse\_ ## fn ## \_stat

一个stat\_parser会从str\_begin到str\_end限定的字符串中取出下一个Statement的所有部分，分析后将其填入stat中，然后返回一个指向剩余字符串的迭代器。所有的stat\_parser实现均在parse\_stat.cpp中。

在分析语句时，如果遇到表达式，则会调用parse\_expr，其实现在parse\_expr.cpp，函数原型如下:

void Parser::parse\_expr(string::const\_iterator str\_begin,

string::const\_iterator str\_end,

size\_t origin,

shared\_ptr<Expression>& expr);

该函数会将从str\_begin到str\_end限定的字符串解析为一个表达式，表达式类型的匹配通过正则按照优先级从高到低进行匹配（在运算符的优先级层面来看，等价于优先级较低的运算符会优先匹配），然后递归地对操作数进行解析，解析完成后，将相关内容填入相应类型的Expression对象实例中，并将其指针赋值给expr。

以上是语法分析部分，语法分析之后，Parser::parse会调用分析得到的一个CompoundStat对象的execute函数，对结果进行计算，执行过程中如果遇到表达式，则调用相应的eval函数进行求值，相关内容可参见“数据结构设计及实现”一节。执行完之后，得到最终结果，控制流返回main函数，然后main函数将结果输出。