

武汉大学计算机学院

本科生实验报告

CMM 解释器系统总体设计与实现

专 业 名 称 : 软件工程

课 程 名 称 : 解释器构造实践

指 导 教 师 一: 李蓉蓉 讲师

学 生 学 号 : 2016302580243

学 生 姓 名 : 李家君

二〇一八年十月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：_____ 日期：_____

1 实验目的和意义

1.1 实验目的

本次实验的内容为 CMM（C 语言的子集）语言的解释器的实现，涉及编译原理的词法、语法、语义等针对性知识，也涉及 C++语言的使用、UML 图的使用、面向对象设计思想和人机交互等软件工程的通用知识。通过本次实验，本人将会对编程有更深入的认识，解决问题的效率也会有大幅提升。同时，本次实验也是对上学期课程“编译原理”的实践补充，有助回忆和巩固课堂知识。

2 实验设计和实施

2.1 实验概述

CMM（C Minus Minus），是 C 语言的子集，其只有 `int` 和 `real` 两个数据类型，条件转移语句仅有 `if` 和 `while` 两种，有两个函数 `read`、`write`，并且不支持指针。

本次实验要实现以 CMM 为源语言的解释器，使用 C++语言编写，具体阶段有词法分析、语法分析、语义分析和解释执行。其中，词法分析为解释器的首要部分，主要功能是分析源程序产生 token 序列供语法分析使用，语法分析将生成语法树，语义分析检查语法树的合法性，解释执行则根据输出程序的结果。

此外，每个阶段都不可避免的有出错处理，编写时本人也是出于用户的角度去考虑问题，尽可能地编写出人性化的错误提示程序。

2.2 词法分析

词法分析是编译的第一阶段。词法分析器的主要任务是读入源程序的输入字符，将它们组成 token，生成并输出一个词法单元序列，这个词法单元序列被输

出到语法分析器进行语法分析。另外，由于词法分析器在编译器中负责读取源程序，因此除了识别词素之外，它还会完成一些其他任务，比如过滤掉源程序中的注释和空白，将编译器生成的错误消息与源程序的位置关联起来等。

2.2.1 基本数据类型

词法分析的主要任务是得出 token 序列。token 是对程序的基本元素，除了自身字符串之外，token 还要记录其所在程序中的位置和类型分类。按照面向对象的思维，一般会把 token 封装成类，但考虑到 token 实际上在生成以后就不会再去修改，为了简便起见，程序统一用 `std::vector<string>` 存储 token，并人为规定每个 token 的数据格式。

```
bool lexicalAnalysis(std::string fileName, std::vector<std::string>& outputs)
{
    // Output format goes like: token1, tokenType1, line1, token2, tokenType2, line2, ...
```

(token 各属性存储格式)

需要说明的是，token 的类型由 enum `tokenType` 表示，代码在 `lexical.h` 中，所以 `tokenType` 只是 enum 类型转成字符串（实际上就是整数字符串），且可以通过以下形式还原为 enum：

```
tokenType type = (tokenType)std::atoi(tokenTypeStr.c_str())
```

(从字符串中还原 tokenType)

`tokenType` 有两种定义思想，第一种是只定义大范围，例如将 `if`、`else`、`while`、`int`、`real`、`read` 和 `write` 都归类为 `keyword`，细分由下一阶段的语法分析在实现；第二种则是在词法阶段就细分。本程序采用的是后者，因为这样定义减轻了语法分析的负担。

```
enum tokenType
{
    IF, ELSE, WHILE, READ, WRITE, INT, REAL, // Keywords
    LPARENT, RPARENT, LBRACKET, RBRACKET, LBRACE, RBRACE, SEMI, COMMA, SCOM, LCOM, RCOM, // Community sites
    PLUS, MINUS, MULT, DIV, ASSIGN, GT, LT, EQ, NEQ, // Binray operators
    NEG, // Unary operators
    IDENTIFER, // Identifers
    INUM, RNUM, // Numbers
    UNIDENTIFIED, UNCLOSESITE, WNUM, // Errors
    NOTYET, // Not get the type yet
};
```

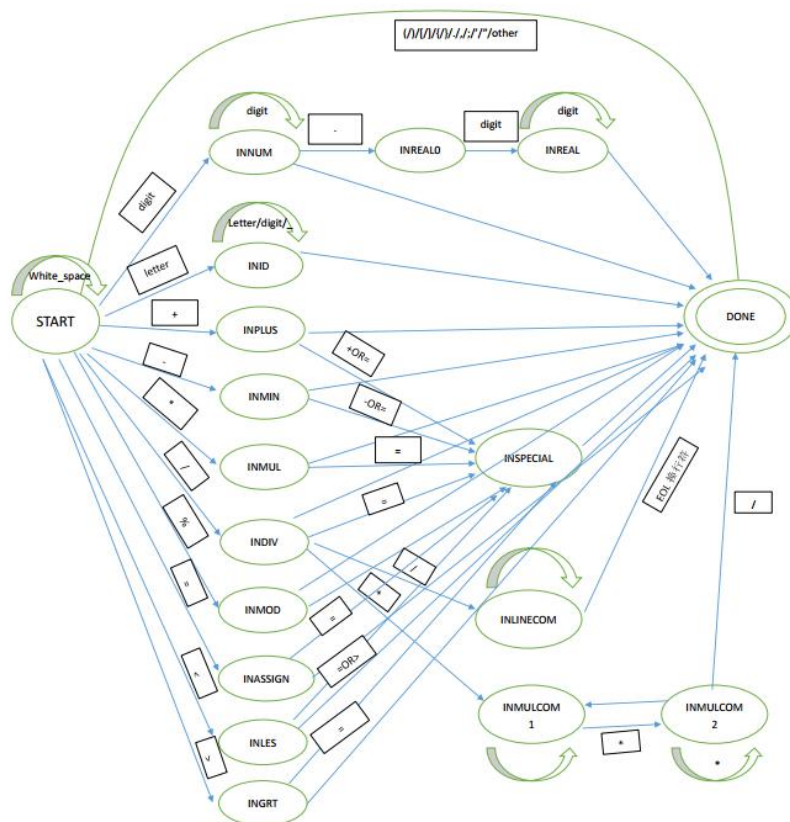
(枚举 tokenType)

2. 2. 2 程序整体逻辑

词法分析对 token 的识别可以由有穷自动机表示，因此用有面向过程特性的语言非常容易实现。主要代码在 `lexical.cpp` 中。

在本程序中，函数 `lexicalAnalysis()` 实现了大部分的识别流程。该函数返回值为 `bool`，若分析出错误则返回 `False`，反之则 `True`，而 `token` 序列通过参数引用得到，这样做可以提示接下来的语法分析阶段前面分析是否出错，若有错误则没必要再进行下一阶段分析。

此外,该函数的输入参数为 `fileName`,意味着函数里会有文件流操作的代码,若从一般的软件工程角度来看,这不符合模块化程序的思想。然而这样做的好处是可以定制化地操作文件流,也不必一次性将源文件全部读入内存再执行词法分析。因为此次实验项目的具体要求已经给定,不必考虑扩展程序,因此在这里采用效率最高的方式实现也是可取的。



(识别 token 的有穷自动机)

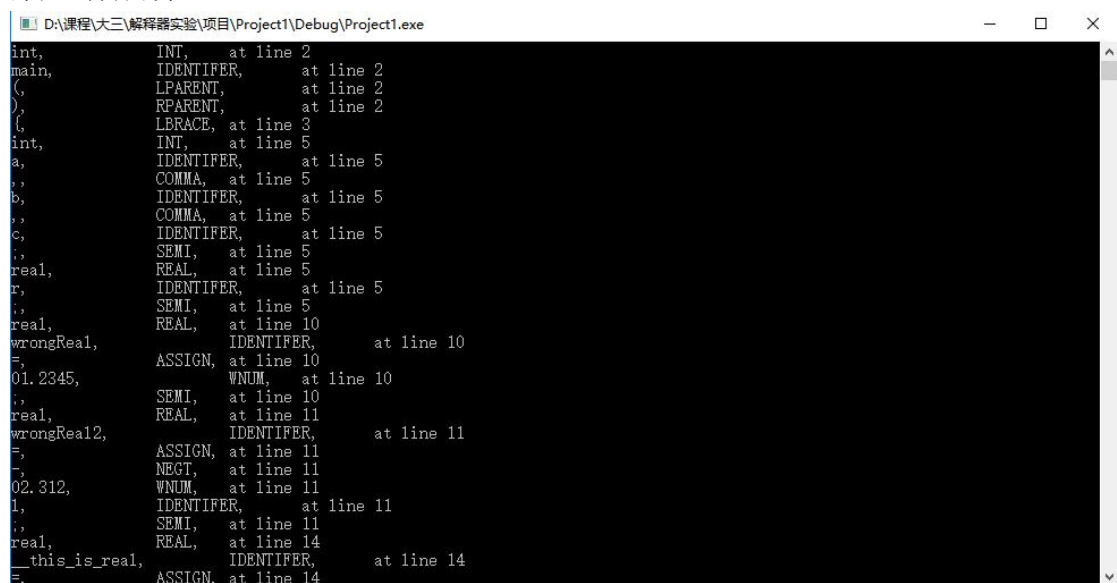
2. 2. 3 出错提示

虽然错误检测并不是词法分析的任务，但本人认为词法分析仍应该尽可能检测出程序的错误。此阶段可以检测的出错主要有三个。第一个是未识别标识符，其判断标准为非数字、下划线和字母打头的字符串，可能是特殊符号或者汉字，都被归类为 `tokenType::unidentified`。第二个是以 0 开头的数字，因为根据 CMM 标准定义，其并不支持八进制数字，这类错误被归为 `tokenType::WNUM`。最后一种是未闭合的多行注释，归为 `tokenType::UNCLOSESITE`。注意，词法分析并不能检测出未闭合的括号，因为涉及到语义，该检测将在生成语法树时发生。

在检测出上述任何一种错误后，`lexicalAnalysis` 都会返回 `false`，意味着将不再执行下一阶段的语法分析。

2. 2. 4 测试

本次词法分析的源代码为 `test.cmm`。书写简单的输出代码即可检测解释器程序是否运行成功。



```
D:\课程\大三\解释器实验\项目\Project1\Debug\Project1.exe
int,          INT,          at line 2
main,         IDENTIFER,    at line 2
(,            LPARENT,     at line 2
),            RPARENT,     at line 2
{,            LBRACE,      at line 3
int,          INT,          at line 5
a,            IDENTIFER,    at line 5
,,            COMMA,       at line 5
b,            IDENTIFER,    at line 5
,,            COMMA,       at line 5
c,            IDENTIFER,    at line 5
,,            SEMI,        at line 5
real,         REAL,        at line 5
r,            IDENTIFER,    at line 5
,,            SEMI,        at line 5
real,         REAL,        at line 10
wrongReal,    IDENTIFER,    at line 10
=,            ASSIGN,      at line 10
01.2345,      WNUM,        at line 10
,,            SEMI,        at line 10
real,         REAL,        at line 11
wrongReal2,   IDENTIFER,    at line 11
=,            ASSIGN,      at line 11
r,            NEG,         at line 11
02.312,       WNUM,        at line 11
l,            IDENTIFER,    at line 11
,,            SEMI,        at line 11
real,         REAL,        at line 14
_this_is_real, IDENTIFER,    at line 14
=,            ASSIGN,      at line 14
```

(部分输出结果，输出格式为 tokenStr tokenType line)

2.3 语法分析

2.3.1 基本数据类型

语法分析的任务是读取词法分析的分词结果来构造语法树，并且报告一些错误。直观地，就会需要一个语法树节点类和表示节点状态的 Enum。但语法树节点形式非常多变，还能存在不确定的子节点个数，一个可能的解决方案是分别定义几个类来表示树节点，另一种方法是设定有限个数的子节点，但用特定的规则来决定所有子节点该怎么样装入。本次实验采用的是后者，只定义 3 个子节点，并且额外定义一个 next 节点专门用于 block 的分支节点，其中的应用规则将在下面介绍。这些节点的使用规则在源码注释中。

```
enum treeNodeStmt
{
    IF_STMT, WHILE_STMT,                // Control statement
    READ_STMT, WRITE_STMT,
    DECLARE_STMT,
    ASSIGN_STMT,
    LBRACE_STMT,                         // New block statement
    EXP_STMT,                           // General expression statement
    ADDITIVE_EXP_STMT, LOGICAL_EXP_STMT, // Note that term expression is an exception
    ...                                 // or additive expression
    VAR_STMT,                           // Variable
    OP_STMT,                            // Operator
    TERM_STMT, FACTOR_STMT,
    NUMBER_STMT,
    NOTYET_STMT,
};
```

(节点状态 Enum)

```
struct treeNode
{
    treeNodeStmt stmt;

    /* ... */
    treeNode* mNext;

    /* ... */
    treeNode* mLeft;
    treeNode* mMiddle;
    treeNode* mRight;

    std::string content;
    tokenType mTokenType;

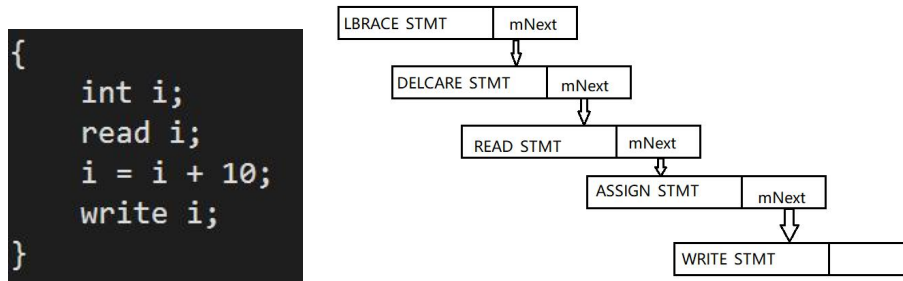
    treeNode() { ... }
    treeNode(treeNodeStmt stmt) { ... }
};
```

(语法树节点类)

以 if 节点为例。这里规定 mLeft 存储 if 条件中的语句节点，mMiddle 存储条件正确时的部分，mRight 存储 else 模块部分（如果有的话），而 content 和

mTokenType 未设置值。

一个特殊的例子是 block 节点（在程序中 treeNodeStmt 为 LBRACE_STMT）。block 节点没有用到 mLeft, mMiddle 和 mRight, 仅用到 mNext 存储 block 的第一个条语句节点, 然后这个语句节点的 mNext 存储下一条语句节点, 以此类推, 形成链表。下图举例表示表示了 block 节点的链接方式:



(一个 block 节点例子和在内存中的表示)

因此可以看到, 每个节点并不一定用到所有的成员变量, 这是因为只用一个类来表示所有类型的节点, 但这种内存的浪费微不足道, 所以这个方案也是可取的。此外, 语法树节点只是存储了必要的信息, 并不是按传统的形式将所有的符号都存储 (如 If 节点没存储小括号和大括号), 这是因为这些符号完全可以从节点类型中推到出来 (每一个 If 节点都有小括号)。

2. 3. 2 程序整体逻辑

本次语法分析采用 LL(1) 自顶向下的分析方式, 主体代码在 parse.cpp。所谓 LL(1) 分析指的是从左到右分析字符串, 并且往前看一个 token。因此在 cpp 文件中用整形 next 表示 token 数组中的下一个。入口函数是 bool syntaxAnalysis(vector<string> input, vector<treeNode*> output), input 为词法分析的结果, output 为本函数输出结果, 且与词法分析入口函数类似, 返回值 bool 表示分析是否遇到错误。

程序主体是多个 parseXXXXStmt() 函数族, 如名字所示, 每一个函数分析对应的语法节点。同样以 If 节点为例讲解。以下是 If 节点的分析程序:

```
treeNode* parseIfStmt()  
{  
    treeNode* node = new treeNode(treeNodeStmt::IF_STMT);  
  
    consumeNextToken(tokenType::IF);  
    consumeNextToken(tokenType::LPARENT);  
    node->mLeft = parseExpStmt();  
    consumeNextToken(tokenType::RPARENT);  
    node->mMiddle = parseStmt();  
  
    // 'else' is an option  
    if(checkNextToken(tokenType::ELSE))  
    {  
        consumeNextToken(tokenType::ELSE);  
        node->mRight = parseStmt();  
    }  
  
    return node;  
}
```


(函数 `parseIfStmt()`)

此函数的整体思想就是按照 if 语句的构造规则进行检查。其中 `consumeNextToken()` 函数验证下一个函数是否与传入的参数相同，相同则增加 `next`，否则给出错误提示，出错处理将在 2.3.3 讨论。

另一个更复杂的例子是对多项式的分析。多项式分析的入口函数为 `parseExpStmt()`。

```
treeNode* parseExpStmt()
{
    // Let's assume it is logical expression
    // LogicalExp => AdditiveExp LogicalOp AdditiveExp | AdditiveExp

    treeNode* leftNode = parseAdditiveExpStmt();

    if(checkNextToken(tokenType::GT) || checkNextToken(tokenType::LT) ||
        checkNextToken(tokenType::EQ) || checkNextToken(tokenType::NEQ))
    {
        treeNode* node = new treeNode(treeNodeStmt::LOGICAL_EXP_STMT);
        node->mLeft = leftNode;
        node->mMiddle = parseLogicalOp();
        node->mRight = parseAdditiveExpStmt();

        return node;
    }

    return leftNode;
}
```

(函数 `parseExpStmt()`)

例如分析表达式 $2 + 3 * 5$ 。程序首先假设这是一个 logical expression，随后若未检测到 logical operator 则其是一个 additive expression，进入函数 `parseAdditiveExpStmt()`：

```
treeNode* parseAdditiveExpStmt()
{
    // AdditiveExp => Term AdditiveOp AdditiveExp | Term

    treeNode* leftNode = parseTermStmt();

    if (checkNextToken(tokenType::PLUS) || checkNextToken(tokenType::MINUS))
    {
        treeNode* node = new treeNode(treeNodeStmt::ADDITIVE_EXP_STMT);
        node->mLeft = leftNode;
        node->mMiddle = parseAdditiveOp();
        node->mRight = parseAdditiveExpStmt();

        return node;
    }

    return leftNode;
}
```

(函数 `parseAdditiveExpStmt()`)

可以看到 additive expression 也是采用类似的思想，先假设这是 term expression，如果检测到 additive operator 再纠正，接下来的函数 `parseTermStmt()` 也类似：

```

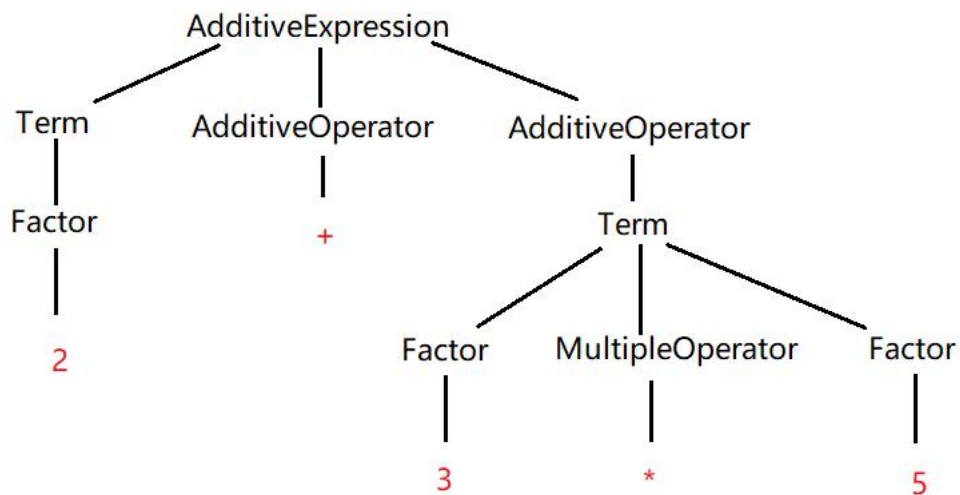
treeNode* parseTermStmt()
{
    // Term => Factor MultiplyOp Term | Factor

    treeNode* leftNode = parseFactor();

```

(函数 parseTermStmt())

这样层层递进的函数实际上也规定了 operator 的优先级，优先级较高的乘号和除号在比加号减号更下层的函数中，表示优先处理。最终表达式的语法树如下：



(表达式分析语法树)

2. 3. 3 出错处理

由于 cmm 语法的简洁性，绝大部分语句都为单语句，以分号结尾，因此出错处理可以设定为对出错语句提示错误并且跳到下一语句继续分析。这样实现起来非常简单，只需过滤掉下一分号前的 token 即可，不再对照代码讲述。

以下是几个常规的错误和对应的提示：

```

6 //---Error 1---
7 real;
8 //-----

```

(错误 1)

```

10 //---Error 2---
11 dadadda
12 int dadada = 0;
13 //-----

```

(错误 2)

```

15 //---Error 3-----
16 arr[1] + ;
17 //-----

```

(错误 3)

```

30 //---Error 4---
31 while(1
32   int j;
33     int jj = 0;
34     jj = jj + 1;
35
36
37
38
39
40
41
42

```

(错误 4)

```

Project1\Debug\Project1.exe
At line7: next token should be IDENTIFER
At line12: next token should be ASSIGN
At line16: next token should be ASSIGN
At line32: next token should be RPARENT

```

(错误提示)

```

Assign Statement
left:
  Variable Statement; content:jj
middle:
  Additive Expression Statement
  left:
    Factor Statement
    left:
      Variable Statement; content:jj
  middle:
    Operator Statement; tokenType:PLUS
  right:
    Factor Statement
    left:
      Number Statement; content:1; tokenType:INUM

```

(错误 4 附近的语法树)

需要说明的是，这样的出错处理与市面上多数编译器不同。对于绝大多数编译器，例如上面提到的错误 4，则会右括号出现以前的所有语句都略过，这样做

也是万不得已的做法，因为这些编译器对应的语言比 cmm 复杂得多，编译器无法有效地猜测用户的出错原因，所有干脆全部略过，让用户改好了再语法分析。

总而言之，出错处理并没有标准答案，应该具体问题具体分析。

2.3.4 测试

程序输出了正确的语法树，说明程序正确。部分结果如下：

```
语法树如下：
Declare Statement; tokenType:INT
left:
  Variable Statement; content:i; tokenType:IDENTIFIER
middle:
  Additive Expression Statement
  left:
    Factor Statement
    left:
      Number Statement; content:2; tokenType:INUM
  middle:
    Operator Statement; tokenType:PLUS
  right:
    Term Statement
    left:
      Factor Statement
      left:
        Number Statement; content:3; tokenType:INUM
      middle:
        Operator Statement; tokenType:MULT
      right:
        Factor Statement
        left:
          Number Statement; content:5; tokenType:INUM

Declare Statement; tokenType:INT
left:
  Variable Statement; content:a; tokenType:IDENTIFIER
```

2.3.5 程序扩展

cmm 的基本语法并不支持 “int a,b,c;” 这样的连续赋值语法，但因为多数都语言支持这种语法，所有本程序在此次稍微做了扩展，以支持这个语法。

实现思路也非常简单，将 declare statement 的所有变量节点用 mNext 连成起来即可，额外的，每个变量的赋值节点将存在该节点的 mMiddle 中。这样是可行的，因为按照先前的规则，变量节点的 mNext 和 mMiddle 都没有使用。

2.4 语义分析+解释执行

语义分析根据语法分析输出的语法树进行进一步解释，可以直接执行，也可生成四元组形式的中间代码。由于本次实验为要求生成中间代码，因此一下的实现是解释完毕直接执行。

2. 4. 1 基本数据类型

该阶段需要三个数据类型，分别是值类型 Value、符号类型 Symbol 和符号表类型 SymbolTable。

Value 代表一个实数或一个整数的数值。注意其包含两个函数。getCastBool() 用于返回转换后的 bool，例如对于 “if(5)” 语句中的 “5” 的 Value，该函数返回 true。getNegativeValue() 用于返回取反的 Value。

```
struct Value
{
    SVType type;
    int mInt;
    double mDouble;

    Value() { ... }

    bool getCastBool() { ... }

    Value getNegativeValue() { ... }
};
```

(Value 类)

Symbol 代表一个变量或者数组。对于数组，每个 Symbol 变量只表示数组的一个元素，而 arrayNext 指向下一个数组元素。arrayInfo 存储数组的信息，例如对于 “int array[5][6][7]”，arrayInfo 存 5 6 7。实际上这是用一维链表存储了多维数组的信息，但也是可取的，因为 cmm 语法不支持指针，而 array、array[5] 这种不完整的表示本身就是指针，因此它们不可能在 cmm 程序中出现，所以可以用一维来存储多维。

除了以上信息之外，还有一个 Symbol* next，指向符号表中同名且在上一个作用域的元素。

```
struct Symbol
{
    SVType type;
    std::string name;
    int level;
    Value value;

    void setType(SVType type) { ... }

    int arrayCount;           // Element count of array
    std::vector<int> arrayInfo; // Demension values of array
    Symbol* arrayNext;        // Next element in the array

    Symbol* next;             // Next element in the symbol table
};
```

(Symbol 类)

SymTable 表示程序运行的符号表，并有删除或添加元素的必要函数。

```

struct SymbolTable
{
    std::vector<Symbol*> symbols;
    Symbol* nonExistSym;

    SymbolTable() { ... }

    Symbol* registerSymbol(Symbol* sym) { ... }

    void deleteSymbols(int level) { ... }

    Symbol* findSymbol(std::string name) { ... }
};

```

(SymbolTable 类)

此外还需要说明 SVType 类型,SV 表示其可表示 Symbol 和 Value 的 type。

```

enum SVType
{
    SV_INT,SV_REAL, // For both symbol and value
    VALUE_TRUE,VALUE_FALSE, VALUE_NONEXIST // For value type only
};

```

(SVType 枚举)

2. 4. 2 程序整体逻辑

实现思路类似于语法分析，程序总入口是函 interpret()，interpretXXX() 函数族分析每一个对应的节点。以解释 If 节点为例。

根据 If 的语法树，可知其 mLeft 节点存条件节点，mMiddle 存 if 分支，mRight 存 else 分支。因此 interpretIf() 主要是根据 mLeft 得出的 Value 确定程序的下一步，代码如下：

```

void interpretIf(int level, treeNode* ifNode)
{
    Value v = interpretExp(level, ifNode->mLeft);

    if(v.type == SVType::VALUE_TRUE || v.getCastBool())
    {
        interpretStmt(level, ifNode->mMiddle);
    }
    else if(ifNode->mRight != NULL)
    {
        interpretStmt(level, ifNode->mRight);
    }
}

```

(interpretIf()函数)

一个特殊的例子是 interpretExp()。根据语法分析，表达式节点可以分为 Logical Expression, Additive Expression, Term Expression 和 Factor 四种情况，没有泛指 Expression 的节点类型，因此 interpretExp() 任务是

决定该节点具体是四种情况中的哪一种，然后交给专门的函数解释。

```
Value interpretExp(int level, treeNode* expNode)
{
    // An expression could be LogicalExp, Term, AdditiveExp or Factor

    switch (expNode->stmt)
    {
        case treeNodeStmt::LOGICAL_EXP_STMT: return interpretLogicalExp(level, expNode);
        case treeNodeStmt::ADDITIVE_EXP_STMT: return interpretAdditiveExp(level, expNode);
        case treeNodeStmt::TERM_STMT: return interpretTerm(level, expNode);
        case treeNodeStmt::FACTOR_STMT: return interpretFactor(level, expNode);
        default: Value v; v.type = SVType::VALUE_NONEXIST; return v;
    }
}
```

(interpretExp()函数)

另一个特殊的例子是 interpretVar()，其功能是将节点表示变量的节点从符号表中找出并返回。对于单个变量，可以直接用前面提到的 SymbolTable::findSymbol() 直接得到。而对于 arr[0][1][2] 这种变量，需要现将多维转换为一维（即计算该变量前面有多少个数组元素），然后在根据数组名字 arr 从符号表中找到链表的首元素，然后顺着二维链表寻找即可。具体实现参考源码。

2. 4. 3 出错处理

此阶段检测的错误主要有数组下标越界、非整数，不完整引用数组，赋值类型不匹配，除数为 0。

数组下标的合法性检测在 interpretDeclare() 与 interpretVar() 中，其实现就是简单通过检查下标的 Value 是否代表整型，和其数值是否在合法范围内。

```
// Lefttest demension
Value value = interpretExp(level, declareNode->mRight);
if(value.type != SVType::SV_INT)
{
    throw std::runtime_error("At line " + std::to_string(declareNode->mRight->line) +
        ": Array index should be integer.");
}
```

(interpretDeclare()中检查最左侧下标是否为整数的部分)

不完整引用数组的检查在 interpretVar() 中，根据前面提到的，Symbol::arrayInfo 存储数组的每个维度，而 arrayInfo.size() 实际就是维数的个数。利用这个就能检查引用的维数是否正确。

```
if(sym->arrayInfo.size() != indexes.size())
{
    throw std::runtime_error("At line " + std::to_string(varNode->line) +
        ": Incorrect array index count.");
}
```

(interpretVar()中检测数组维数是否正确的部分，indexes 是变量节点得到的维度信息)

类型是否匹配检查在 interpretAssign() 中。根据 cmm 语法，real 类型数值直接赋值到 int 类型变量是不允许的，因为会损失小数点的数据，引起误差。但反过来是允许的，且 real 小数部分自动补零（这里利用了 C++ 语法的类型转换特性直接实现了）。

```

Value v = interpretExp(level, assignNode->mMiddle);
Symbol* sym = interpretVar(level, assignNode->mLeft);

if(sym->type == v.type)
{
    sym->value = v;
}
else if (sym->type == SVType::SV_REAL && v.type == SVType::SV_INT)
{
    sym->value.mInt = v.mInt;
    sym->value.mDouble = v.mDouble;
}
else if(sym->type == SVType::SV_REAL && v.type == SVType::SV_INT) // Safety cast
{
    sym->value.mDouble = v.mInt;
}
else
{
    throw std::runtime_error("At line " + std::to_string(assignNode->line) +
        ": Unsupport cast." );
}

```

(interpretAssign())的类型检查部分)

除 0 检查在 interpretTerm() 中。

```

if(!IS_DOUBLE_NOT_ZERO(right.mDouble))
{
    throw std::runtime_error("At line " + std::to_string(termExpNode->line) +
        ": Divider must be non-zero.");
}

if (right.mInt == 0)
{
    throw std::runtime_error("At line " + std::to_string(termExpNode->line) +
        ": Divider must be non-zero.");
}

```

(interpretTerm()中检测除数是否为 0 部分)

需要注意的是，C++的 double 类型因为精度有限，两个值相当的 double 相减并不能得到 0，所以这里实际上给了一个可承受误差范围，相减值足够小就认为其为 0。

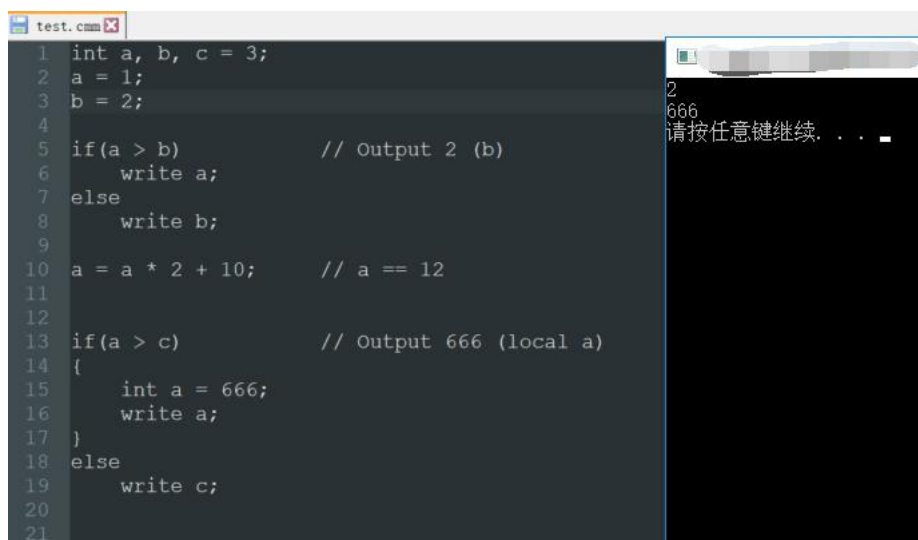
```

#define MIN_VALUE 1e-8
#define IS_DOUBLE_NOT_ZERO(d) (std::abs(d) > MIN_VALUE)

```

2. 4. 4 运行测试

简单的测试程序和输出结果如下所示：



```
test.cmm
1  int a, b, c = 3;
2  a = 1;
3  b = 2;
4
5  if(a > b)          // Output 2 (b)
6      write a;
7  else
8      write b;
9
10 a = a * 2 + 10;    // a == 12
11
12
13 if(a > c)          // Output 666 (local a)
14 {
15     int a = 666;
16     write a;
17 }
18 else
19     write c;
20
21
```

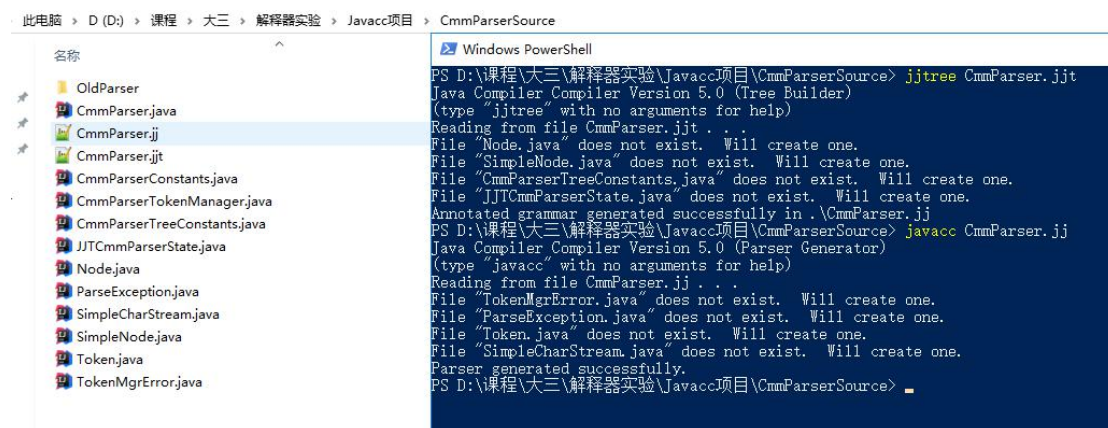
2
666
请按任意键继续. . .

可见，程序正常运行。

2.5 Javacc

2. 5. 1 Javacc 使用方法

Javacc 是基于 Java 的自 LL 解析生成器，使用方法是利用 .jjt 文件生成 .jj 文件和部分 Java 文件，然后根据 .jj 最终生成完整的 Java 代码。下图显示了由 .jjt 文件生成最终完整的 Java 代码的过程：



2. 5. 2 JJTree 程序设计

利用 SKIP 的语法可以很好地过滤注释。

```

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| < "/" (~[ "\n", "\r" ])*           // Filter single line
  (
    "\n"
  | "\r"
  | "\r\n"
  ) >
| < "/" (~[ "*" ])* "*"           // Filter multiple lines
  (
    ~[ "/" ] (~[ "*" ])* "*"
  )*
  "/" >
}

```

接着是声明 Token，Javacc 通过匹配 Token 来实现 LL 语法的分析。本程序没有在 Option 中指示向解析时前看几个 Token，因此默认为向前看一个。

```

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER : < LETTER > (( < LETTER > | < DIGIT > | "null")* ( < LETTER > | < DIGIT > ))? >
| < #LETTER : ["a"- "z", "A"- "Z"] >
| < #DIGIT : ["0"- "9"] >
}

```

```

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL : < DECIMAL_LITERAL >>
| < REAL_LITERAL : ( < DECIMAL_LITERAL > ) ( "." ( ["0"- "9"] )+ )? >
| < #DECIMAL_LITERAL : "0" | ( ["1"- "9"] ( ["0"- "9"] )* ) >
}

```

```

TOKEN : /* KEYWORDS */
{
  < IF : "if" >
| < ELSE : "else" >
| < WHILE : "while" >
| < READ : "read" >
| < WRITE : "write" >
| < INT : "int" >
| < REAL : "real" >
}

```

接下来是定义函数来描述 Cmm 语法，思路与前面介绍的语法分析类似。首先以匹配 If 语句来介绍。

```

void ifStmt():
{
    String value = null;
}
{
    <IF> <LC> value=condition() <RC>
    {
        FourCode ifCode = new FourCode("jmp", value, null, null);
        fourCodes.add(ifCode);
    }
    blockStmt()
    { ifCode.setForth(fourCodes.size() + ""); }

    // Else part is an option
    (
        <ELSE>
        {
            FourCode elseCode = new FourCode("jmp", null, null, null);
            fourCodes.add(elseCode);
            ifCode.setForth(fourCodes.size()+ "");
        }
        blockStmt()
        { elseCode.setForth(fourCodes.size()+ ""); }
    )?
}

```

代码主要做的就是匹配源码中的各个 Token，例如 “<IF><LC>value = condition() <RC>” 就是匹配 “if(condition)”，然后生成四元式(jmp, value, null, line)，其中 line 指向是 if 语句块执行完后的下一个四元式，这个四元式的意思是“如果 value 解析出为 false 的话，下一条执行的语句就为 line 指向的语句”。所有四元式的说明在 FourCode.java 中。

另一个典型的例子是 whileStmt()。

```

void whilestmt():
{
    String value = null; int line = -1;
}
{
    <WHILE> <LC> value = condition() <RC>
    {
        line = fourCodes.size();
        FourCode jumpCode = new FourCode("jmp", value, null, null);
        fourCodes.add(jumpCode);
    }
    blockStmt()
    {
        jumpCode.setForth(String.valueOf(fourCodes.size()+1));
        fourCodes.add(new FourCode("jmp", null, null, line+""));
    }
}

```

类似于 IfStmt()，在 while 语句的开头也是先生成条件跳转四元式，但为了构成循环，还要最后一行加上一条无条件跳转(jmp, null, null, line)，其中 line 指向 while 的第一句，即条件跳转四元式。

一个特殊的例子是函数 Variable()。

```
String Variable():
{
    Token token = null;
    String index = null, name = null;
}
{
    token = <IDENTIFIER> (<LM> index = expression() <RM>)?
    {
        if(index == null)    name = token.image;
        else                 name = token.image + "[" + index + "]";
    }

    { return name; }
}
```

此函数用于解析变量，而变量可能为单值变量或数组变量。若为前者则函数返回变量名；若解析的是数组变量，则返回“variableName[index]”的形式。

2. 5. 3 四元式解释器

上一小节介绍的 JJTree 程序仅能用来生成四元式，而解释执行这些四元式需要额外编写 Java 代码。此次实验编写解释执行代码主要在 RunCommand.java 中，因为该内容和语义分析类似，这里就不再赘述。

2. 5. 4 测试

如果直接在命令行执行 Java 代码可能会造成找不到包的错误，因此这里首先用 IDEA 创建了一个空白 Java 项目，在其中创建名为 test 的包，然后把 Javacc 生成的所有 java 代码拖入 test 中执行。

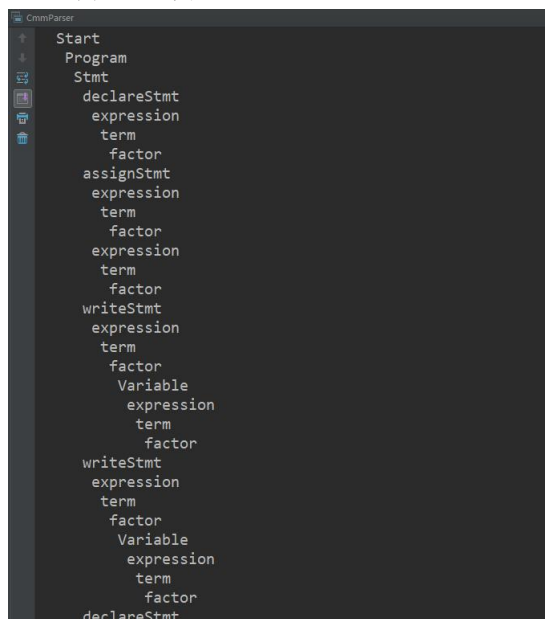
测试文件 test.cmm 如下：

```
/*
    Array Definition
*/
int arr[10];
arr[2] = 666;
write(arr[0]);    // Output 0
write(arr[2]);    // Output 666

int a = 3;
while(a < 10)
{ a = a + 1; }

if(a < 10)
{
    write (111);
}
else
{
    if(a <> 10)
    {
        write (222);
    }
    else
    {
        int a = 20;
        write (a+2);    // Output 22
    }
    write (333);    // Output 333
}
```

生成得语法树:



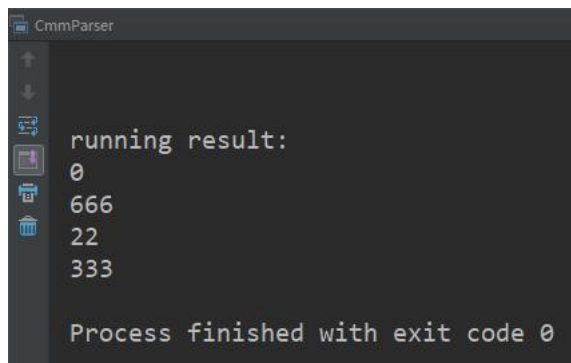
运行生成的四元式如下:

```
0 : (int, null, 10, arr)
1 : (=, 666, null, arr[2])
2 : (write, null, null, arr[0])
3 : (write, null, null, arr[2])
4 : (int, null, null, a)
5 : (=, 3, null, a)
6 : (<, a, 10, *temp1)
7 : (jmp, *temp1, null, 13)
8 : (in, null, null, null)
9 : (+, a, 1, *temp2)
10 : (=, *temp2, null, a)
11 : (out, null, null, null)
12 : (jmp, null, null, 6)
13 : (<, a, 10, *temp3)
14 : (jmp, *temp3, null, 19)
15 : (in, null, null, null)
16 : (write, null, null, 111)
17 : (out, null, null, null)
18 : (jmp, null, null, 34)
19 : (in, null, null, null)
20 : (<>, a, 10, *temp4)
21 : (jmp, *temp4, null, 26)
22 : (in, null, null, null)
23 : (write, null, null, 222)
24 : (out, null, null, null)
25 : (jmp, null, null, 32)
26 : (in, null, null, null)
27 : (int, null, null, a)
28 : (=, 20, null, a)
29 : (+, a, 2, *temp5)
30 : (write, null, null, *temp5)
31 : (out, null, null, null)
32 : (write, null, null, 333)
33 : (out, null, null, null)
```

其中*temp 为计算过程中定义的临时变量，具体生成规则参考

SymbolTable.java。

执行上述四元式结果如下：



```
CmmParser
↑
↓
running result:
0
666
22
333

Process finished with exit code 0
```

可见程序正常执行。

3 实验心得

3.1 词法分析心得

词法分析是一系列实验的基础，接下来的两次实验都是基于此实验而进行的。因为没有编写解释器的先前经历，所以本人在刚开始确定数据结构的时候纠结了很久，参考了很多网上的例子但都不能做出决定。最终把语法分析的内容也温习一遍才确定下所有数据结构，合适的语法结构让我在做下一节实验时非常顺利，可见做任何事在开头阶段认真把握每一个细节是值得的。

3.1 语法分析心得

语法分析中本人采用的一个主要策略是“牺牲一定的空间以换取便利”，例如只定义一个类来表示所有类型的语法树节点，这是一种比较灵活的做法，但在工作中这样做显然是不合适的，不符合软件工程中“单一职责”的原则，在大型代码环境中不易 Debug，因此在以后工作中要避免这种做法。

3.1 语义分析心得

语义分析因为有前两次实验的铺垫，实现起来相对简单，此次实验也为本人巩固了符号表、代码反填等内容。

3.1 Javacc 使用心得

个人认为 Javacc 是非常强大的 LL 语义的解释器生成器，能极大地简化开发者的工作，同样要实现 cmm 的解释器，比起词法语法实验的上千行代码，利用 Javacc 来实现则用了不到 400 行，并且思路更加清晰。此外，生成的四元式中间代码也利于未来的代码优化，因为许多代码优化的方法都是针对四元式而不是源程序的。

