# Toureter Interpreter

——**Interpretive Execution Design Document**

**Written by Xiaoran Zhan**

# Produced by Toureter Group

**All Rights Reserved, Copy Reserved**

目录

[Toureter Interpreter 0](#_Toc345295303)

[Produced by Toureter Group 0](#_Toc345295304)

[1 整体概述 3](#_Toc345295305)

[2 实现概念介绍 3](#_Toc345295306)

[2.1 语法分析对象介绍 3](#_Toc345295307)

[2.1.1 CMM语言简介 3](#_Toc345295308)

[2.3 总体流程图 5](#_Toc345295309)

[2.4拓展CMM语法规则总结 6](#_Toc345295310)

[2.4.1 程序 6](#_Toc345295311)

[2.5 基本逻辑解释 8](#_Toc345295312)

[2.5.1 声明（定义）语句逻辑 8](#_Toc345295313)

[2.5.2 赋值语句逻辑 9](#_Toc345295314)

[2.5.3 循环语句逻辑 10](#_Toc345295315)

[2.5.4 条件语句 11](#_Toc345295316)

[2.5.5 读语句 12](#_Toc345295317)

[2.5.6 写语句 13](#_Toc345295318)

[2.5.7 计算表达式、分表达式、子表达式的值 13](#_Toc345295319)

[3 实现方法介绍 13](#_Toc345295320)

[3.1 所需结构 13](#_Toc345295321)

[3.1.1 符号类结构—超类 13](#_Toc345295322)

[3.1.2四种符号类结构 14](#_Toc345295323)

[3.1.3 符号表结构 15](#_Toc345295324)

[3.1.4 四元式类结构 17](#_Toc345295325)

[} 21](#_Toc345295326)

[3.1.5 设置下一个Token，移动token的方法 21](#_Toc345295327)

[3.1.6 在符号表中查找特定符号并返回该符号的地址 21](#_Toc345295328)

[3.1.7 在符号表中查找元素 23](#_Toc345295329)

[3.1.8 用于计算表达式值的两个函数 23](#_Toc345295330)

[3.2 具体实现 25](#_Toc345295331)

[3.2.1 分析入口 25](#_Toc345295332)

[3.2.2 程序由类和结构体构成 26](#_Toc345295333)

[3.2.3 单条语句 27](#_Toc345295334)

[3.2.4 语句序列 28](#_Toc345295335)

[3.2.5 声明语句 28](#_Toc345295336)

[3.2.6 表达式 29](#_Toc345295337)

[3.2.7 运算表达式 30](#_Toc345295338)

[3.2.8 分表达式 31](#_Toc345295339)

[3.2.9 返回语句 31](#_Toc345295340)

[3.2.10 循环语句 32](#_Toc345295341)

[3.2.11 条件语句 32](#_Toc345295342)

[3.2.12 布尔表达式 33](#_Toc345295343)

[3.2.13 读语句 35](#_Toc345295344)

[3.2.14 写语句 35](#_Toc345295345)

[3.2.15 赋值语句 37](#_Toc345295346)

[4 实例展示 38](#_Toc345295347)

[4.1 无参数类展示 38](#_Toc345295348)

[4.2 有参数类展示 40](#_Toc345295349)

# 1 整体概述

Toureter Interpreter解释执行模块的主要功能是对语法树进行遍历,分析语义、生成中间代码，同时用解释器执行中间代码。

# 2 实现概念介绍

## 2.1 拓展CMM语法简介

词法分析的对象是CMM语言的源程序，源程序的叙述如下：

CMM语言为C语言的一个子集：

* + 语言结构：顺序结构（赋值语句、输入、输出）、选择语句（if-else）、循环结构（while）。这些语句结构和C语言的结构一样，允许嵌套。
  + 表达式局限于关系表达式和算术表达式，运算的优先级为：算术运算、关系运算，并服从左结合规则。
  + 算术表达式包括整数和实数上的运算、变量以及“（）”、“\*”、“+”、“-”、“/”，运算符的优先级顺序为：“（）”大于“\*”和“/”大于“+”和“-”。
  + 关系运算符包括：“<”、“==”、“<>”。
  + 一条语句以“；”结束；程序由一条语句或者由“{”和“}”嵌套表达的复合语句。
  + 支持多行注释（使用“/\*”和“ \*/ ”）
  + 支持数组运算，数组的下标必须是正整数，使用“[”和“]”表示数组下标。
  + 变量的使用之前需要先声明，声明的方式和C语言一样。

|  |  |  |
| --- | --- | --- |
| 保留字 | 特殊符号 | 其他 |
| if | + | 十进制的整数与实数 |
| else | - |
| while | \* |
| read | / | 标识符（由数字、字母和下划线组成的串，但必须以字母开头、且不能以下划线结尾的串） |
| write | = |
| int | < |
| real | == |  |
| struct | <> |  |
| class | ( |  |
|  | ) |  |
|  | ; |  |
|  | { |  |
|  | } |  |
|  | /\* |  |
|  | \*/ |  |
|  | [ |  |
|  | ] |  |
|  | /\* |  |
|  | \*/ |  |
|  | && |  |
|  | || |  |
|  | ! |  |

经过词法分析，得到的符号类型简介如下：

|  |  |
| --- | --- |
| 符号类型 | 符号 |
| Keyword | if，else，while，int，real，read，write |
| Sign | +，-，\*，/，/\*，\*/，=，==，<，<>，[，]，(，)，{，}，;，,， |
| int | 十进制整数，由数字串组成 |
| real | 十进制实数，由数字和小数点组成 |
| Identifier | 由字母开头，由数字、字母和\_组成，并且不以下划线结尾的字符串 |

## 2.3 总体流程图

词法分析的总体流程图如下图所示：

## 2.4拓展CMM语法规则总结

**注意: 大写字母为非终结符(Nonterminal), 小写为终结符(Terminal),有颜色的字体为保留字或者运算符.语法用BNF范式表示**

### 2.4.1 程序

程序由class和struct组成

Program ->( class|struct)\*

* + 1. **类**

CLASS-> class identifier { (DECLARE\_STMT |FUNCTION)\* } //类定义由ID和声明语句,函数组合而成

* + 1. **结构体**

STRUCT-> struct identifier { (DECLARE\_STMT)\* } //结构体由ID和声明语句组成

* + 1. **函数**

FUNCTION-> TYPE identifier (( DECLARE\_STMT)\*) STMT\_SEQUENCE

* + 1. **语句序列**

STMT\_SEQUENCE -> { STATEMENT}\* // 语句序列由多条语句组成

* + 1. **语句**

STATEMENT-> ASSGIN\_STMT|

WRITE\_STMT |

READ\_STMT |

IF\_STMT |

WHILE\_STMT |

DECLARE\_STMT|

CALL\_STMT

* + 1. **循环语句**

WHILE\_STMT -> while ( EXPR\* ) ({ STMT\_SEQUENCE})|(STATEMENT) //while语句

* + 1. **赋值语句**

ASSGIN\_STMT -> identifier = EXPR

identifier -> identifier (“[“ EXPR“]” )\*

* + 1. **条件语句**

If\_STMT -> if ( EXPR\* ) ({ STMT\_SEQUENCE})|(STMT) (else ({STMT\_SEQUENCE})|(STATEMENT))?

* + 1. **读语句**

READ\_STMT -> read identifier; //读语句

* + 1. **写语句**

WRITE\_STMT -> write EXPR; //写语句

* + 1. **表达式语句**

EXPR->LOGIC\_AND\_EXPR

LOGIC\_AND\_EXPR -> LOGIC\_OR\_EXPR (&& LOGIC\_OR\_EPXR)?

LOGIC\_OR\_EXPR ->LOGIC\_NOT\_EXPR (|| LOGIC\_NOT\_EXPR)?

LOGIC\_NOT\_EXPR ->(!)? LOGIC\_ELEMENT\_EXPR

LOGIC\_ELEMENT\_EXPR-> true|false|EQUALITY\_EXPR

EQUALITY\_EXPR -> ARITHM\_EXPR (<|<=|>|>=|==|<> ARITHM\_EXPR)?

ARTHM\_EXPR ->TERM\_EXPR (+|- TERM\_EXPR)?

TERM\_EXPR ->UNARY\_EXPR (\*|/|% UNARY\_EXPR)?

UNARY\_EXPR ->(-)? ELMENT

ELMENT -> const|CALL\_STMT|identifier|(EXPR)

## 2.5 基本逻辑解释

### 2.5.1 声明（定义）语句逻辑

从根节点进行扫描

下一个token是不是“[”

搜索数组符号表，查看是否有当前level的ident定义过

搜索普通变量符号表，查看是否有当前level的ident定义过

是否定义过？

是

否

忽略此次定义，并报错

将该ident连同后面扫描得到的值加入到符号表中

是

否

多个变量声明要考虑“，”的情况，将，作为每一次定义结束的判断条件，如果有“，”则说明新的变量声明，进行相同的操作即可。

### 2.5.2 赋值语句逻辑

扫描到ident，暂时存储

否

搜索普通变量符号表，查看是否有当前level的ident定义过

下一个token是不是“[”

是

获取下一个token，并且一同与ident在数组符号表中搜索

是否定义过？

否

是

将=后面的值赋值给相应的符号表并进行保存

报错，提醒用户越界或未定义的符号

### 2.5.3 循环语句逻辑

扫描“(”后直到条件符号位置，并进行保存

获取下一个token，并进行保存，这个token是条件比较符号

扫描“)”前，并将扫描结果进行保存进行保存，并保存此时ptr的结果

计算并保存while循环结束的ptr的位置，以防止while循环一次都不执行带来的后果

是

根据保存得知计算表达式是否成立

否

退出

While循环执行的成败在于首先判断while结束的地址，之后保存相关的信息进行循环计算，如果布尔表达式成立则执行，如果不成立则调到之前保存的ptr处进行执行。

### 2.5.4 条件语句

计算并保存布尔表达式中的值和比较符号

根据保存得知比较表达式是否成立

是

执行if语句

否

根据下一个token判断是否有else语句

跳过所有if语句的内容

否

是

根据下一个token判断是否有else语句

跳过所有else语句

否

是

结束

执行else语句

If语句执行的关键在于如何跳过不执行的语句，并且判断在没有else的情况下的解释执行过程。

### 2.5.5 读语句

保存read()括号中的内容

读取输入框中的内容

解析read()括号内的内容

否

在符号表中检查是否存在

报错，并提示用户进行修改

是

解析read()括号内的内容并在符号表中检查是否存在

### 2.5.6 写语句

保存write()括号中的内容

否

在符号表中检查是否存在

报错，程序终止，提醒用户

是

输出操作

### 2.5.7 计算表达式、分表达式、子表达式的值

判断现在计算的对象是int还是real，并选择不同的函数进行计算

再调用结束的时候统一进行计算

# 3 实现方法介绍

## 3.1 所需结构

### 3.1.1 符号类结构—超类

internal enum SymbolType { VOID, STRUCT, ARRAY, FUNCTION, BOOLEAN, INT, REAL }

internal class Symbol

{

private string m\_offset = "";

public string Name { get; set; }

public SymbolType Type{ get; set; }

public string Address { get { return this.m\_offset; } set { this.m\_offset = value; } }

public GrammerTreeNode OrginNode { get; set; }

}

### 3.1.2四种符号类结构

internal class FunctionSymbol : Symbol {

private List<Symbol> m\_params = new List<Symbol>();

public SymbolType ReturnType { get { return ReturnValue.Type;} }

public Symbol ReturnValue { get; set; }

public List<Symbol> Parameter { get { return m\_params; } set { this.m\_params = value; } }

public FunctionSymbol() { }

}

internal class VarSymbol : Symbol {

public GrammerTreeNode InitValueExpr { get; set; }

}

internal class StructSymbol : Symbol

{

private List<Symbol> m\_children = new List<Symbol>();

public string StructName { get; set; }

public List<Symbol> Children

{

get { return this.m\_children; }

set { this.m\_children = value; }

}

public StructSymbol() {

}

public int GetIndexByChildName(string name)

{

return this.m\_children.IndexOf(m\_children.Find(c => c.Name == name));

}

public Symbol GetSymbolByName(string name)

{

return this.m\_children.Find(c => c.Name == name);

}

}

internal class ArraySymbol : Symbol {

private List<int> m\_dimensionSize=new List<int>();

public SymbolType ArrayType { get; set; }

public int Dimension { get; set; }

public Symbol Element { get; set; }

}

### 3.1.3 符号表结构

internal sealed class SymbolTable:Dictionary<string,Symbol> {

private const string ENTRANCE = "Main";

private SymbolTable m\_subTable;

private SymbolTable m\_supTable;

private FunctionSymbol m\_entrance;

public SymbolTable SupTable {

get { return this.m\_supTable; }

set { this.m\_supTable = value; }

}

public SymbolTable SubTable {

get { return this.m\_subTable; }

set { this.m\_subTable = value; }

}

public FunctionSymbol Entrance { get { return this.m\_entrance; } }

public SymbolTable() {

}

public void Add(Symbol symbol){

Add(symbol.Name, symbol);

}

public new void Add(string name,Symbol symbol) {

if (!this.ContainsInThisScope(name))

base.Add(name, symbol);

else

throw new DuplicatedDeclareException(symbol.OrginNode.Children[1].TerminalType);

}

public void AddFunction(FunctionSymbol funcSymbol) {

this.Add(funcSymbol);

if (funcSymbol.Name == ENTRANCE)

this.m\_entrance = funcSymbol;

}

public bool Contains(string name) {

if (this.ContainsKey(name))

return true;

else if (this.m\_supTable != null)

return this.m\_supTable.Contains(name);

else return false;

}

public Symbol Find(Token token) {

if (this.ContainsKey(token.Image))

return this[token.Image];

else if (this.m\_supTable != null)

return this.m\_supTable.Find(token);

else

throw new UnDeclaredTypeException(token);

}

public FunctionSymbol GetFunctionSymbol(Token functionName, List<Token> paramTokens,List<Symbol> paramSymbols) {

if (this.ContainsKey(functionName.Image))

{

FunctionSymbol funSymbol = (FunctionSymbol)this[functionName.Image];

if (paramTokens.Count() != funSymbol.Parameter.Count)

throw new UnMatchedParametersException(functionName, paramTokens);

for (int i = 0; i < paramSymbols.Count; i++)

{

if (paramSymbols[i].Type != funSymbol.Parameter[i].Type)

throw new UnMatchedParametersException(functionName, paramTokens[i]);

}

return funSymbol;

}

else

throw new UnDeclaredTypeException(functionName);

}

public Symbol GetMembeSymbol(string structTypeName, GrammerTreeNode memberNode, out int offset) {

if (this.ContainsKey(structTypeName))

{

StructSymbol symbol =(StructSymbol) this[structTypeName];

Symbol memberSymbol = null;

if ((memberSymbol = symbol.Children.Find(c => c.Name == memberNode.TerminalType.Image && c.Type == SemanticAnalyzer.CastNodeToSymbolType(memberNode))) == null)

throw new UnMatchedMemberException(memberNode.TerminalType, symbol);

else

{

offset = symbol.Children.IndexOf(memberSymbol);

return memberSymbol;

}

}

else if (this.m\_supTable != null)

{

return this.m\_supTable.GetMembeSymbol(structTypeName, memberNode, out offset);

}

else

throw new Exception(string.Concat("找不到定义为", structTypeName, "的类型,异常在FindDefinition中发生"));

}

public FunctionSymbol GetEntranceFncSymbol()

{

if (this.Entrance == null)

if (this.m\_supTable == null)

throw new SemanticException("缺少Main函数, 没有找到代码入口");

else

return this.m\_supTable.GetEntranceFncSymbol();

else

return this.Entrance;

}

public bool ContainsInThisScope(string name) {

return (base.ContainsKey(name));

}

public Symbol FindInThisScopr(string name) {

if (this.ContainsInThisScope(name))

return this[name];

else return null;

}

}

### 3.1.4 四元式类结构

internal sealed class Quaternion

{

#region Quaternion command

public const string MALLOC = "MALLOC";

public const string COPY = "COPY";

public const string HOLD = "HOLD";

public const string PUSH = "PUSH";

public const string LOAD = "LOAD";

public const string OFFSET = "OFFSET";

public const string ADD = "ADD";

public const string SUB = "SUB";

public const string MUL = "MUL";

public const string DIV = "DIV";

public const string MOD = "MOD";

public const string GT = "GT";

public const string LT = "LT";

public const string GOE = "GOE";

public const string LOE="LOE";

public const string EQ = "EQ";

public const string NEQ = "NEQ";

public const string AND = "AND";

public const string OR = "OR";

public const string XOR = "XOR";

public const string MINUS = "MIS";

public const string NOT = "NOT";

public const string NEXT = "NEXT";

public const string JUMP = "JUMP";

#endregion

#region quaternion const

public const string OBJ = "OBJ";

public const string LIS = "LIS";

public const string INT = "INT";

public const string REAL = "REAL";

public const string BYVAL = "VAL";

public const string BYREF = "REF";

public const string STACK = "STACK";

public const string EAX = "EAX";

public const string EBX = "EBX";

public const string ECX = "ECX";

public const string EDX = "EDX";

public const string PC = "PC";

public const string EBP = "EBP";

#endregion

public const string STK\_ADDR\_PREFIX = "STK\_";

public const string STK\_ADDR\_SEPERATOR = "\_";

public const string STK\_BOTTOM = "STK\_0";

public const int REG\_COUNT = 5;

#region debug info region

public const string DEBUG\_HEAD = "//";

public const string DEBUG\_varDeclareGenerate = "";

public const string DEBUG\_functionGenerate = "";

public const string DEBUG\_generateTypeCheckAndConvert = "";

public const string DEBUG\_generateExpression = "";

public const string DEBUG\_generateCalcElementAddress = "";

public const string DEBUG\_saveContext = "";

public const string DEBUG\_convertVarType = "";

public const string DEBUG\_CastTokenTypeToSymbolType = "";

private static string debug\_Address = STK\_BOTTOM;

#endregion

public readonly static string[] LOGI\_OPERATORS =new string[] { AND, OR, XOR };

public readonly static string[] COMP\_OPERATORS = new string[] { EQ, NEQ, GT, LT, GOE, LOE };

public readonly static string[] NUM\_OPERATORS = new string[] { ADD, SUB, MUL, DIV, MOD };

private const string SPACE = " ";

public enum AddressType{StackAddr,RegisterAddr,CodeAddr}

public string Value { get; private set; }

public Quaternion(params string[] quaters) {

this.Value = quaters[0];

for (int i = 1; i < quaters.Length; i++) {

this.Value=string.Concat(this.Value, SPACE, quaters[i]);

}

if (quaters[0] == Quaternion.MALLOC)

this.Value=string.Concat(this.Value, SPACE, DEBUG\_HEAD, debug\_Address);

}

public static string AddrIncrease(ref string address) {

if (address.StartsWith(STK\_ADDR\_PREFIX))

{

address=string.Concat(STK\_ADDR\_PREFIX, Convert.ToInt32(address.Split(STK\_ADDR\_SEPERATOR.ToArray())[1])+1);

debug\_Address = address;

return string.Concat(STK\_ADDR\_PREFIX, Convert.ToInt32(address.Split(STK\_ADDR\_SEPERATOR.ToArray())[1])-1);

}

else if (address == EDX)

return ECX;

else

return "0";

public static string CreateStackAddr(int offset) {

return string.Concat("STK\_", offset);

}

public static AddressType GetAddrTypeFromAddr(string addr) {

if (addr.StartsWith(STK\_ADDR\_PREFIX)|| addr=="")

return AddressType.StackAddr;

else if (new string[] { EAX, EBX, ECX, EDX }.Contains(addr))

return AddressType.RegisterAddr;

else

return AddressType.CodeAddr;

}

public static string GetQuaternionTypeFromSymbolType(SymbolType type) {

switch (type) {

case SymbolType.BOOLEAN:

case SymbolType.INT: return INT;

case SymbolType.REAL: return REAL;

case SymbolType.ARRAY:

case SymbolType.STRUCT: return LIS;

default: //TODO: DELETE

throw new Exception("卧槽,GetQuaternionTypeFromSymbolType接收到错误的参数了");

}

}

public static string GetUnaryOperatorByOperatorNode(GrammerTreeNode operatorNode) {

switch (operatorNode.TerminalType.Kind) {

case TokenKind.MINUS: return Quaternion.MINUS;

case TokenKind.NOT: return Quaternion.NOT;

default: //TODO:DELETE

throw new Exception("卧槽,操作符错误!快debug一下");

}

}

public static string GetBinoOperatorByOperatorNode(GrammerTreeNode operatorNode) {

switch (operatorNode.TerminalType.Kind)

{

case TokenKind.AND: return Quaternion.AND;

case TokenKind.OR: return Quaternion.OR;

case TokenKind.PLUS: return Quaternion.ADD;

case TokenKind.MINUS: return Quaternion.SUB;

case TokenKind.MOD: return Quaternion.MOD;

case TokenKind.MULTIPLY: return Quaternion.MUL;

case TokenKind.DIVIDE: return Quaternion.DIV;

case TokenKind.EQ: return Quaternion.EQ;

case TokenKind.NEQ: return Quaternion.NEQ;

case TokenKind.GOE: return Quaternion.GOE;

case TokenKind.LOE: return Quaternion.LOE;

case TokenKind.LT: return Quaternion.LT;

case TokenKind.GT: return Quaternion.GT;

default://TODO DELETE

throw new Exception("卧槽,操作符错误!快debug一下");

}

}

}

### 3.1.5 作用域的实现

public Symbol Find(Token token) { //向上遍历,查看父表中有无该元素

if (this.ContainsKey(token.Image))

return this[token.Image];

else if (this.m\_supTable != null)

return this.m\_supTable.Find(token);

else

throw new UnDeclaredTypeException(token);

}

### 3.1.6 在符号表中查找特定符号并返回该符号的地址

public Symbol Find(Token token) {

if (this.ContainsKey(token.Image))

return this[token.Image];

else if (this.m\_supTable != null)

return this.m\_supTable.Find(token);

else

throw new UnDeclaredTypeException(token);

}

public FunctionSymbol GetFunctionSymbol(Token functionName, List<Token> paramTokens,List<Symbol> paramSymbols) {

if (this.ContainsKey(functionName.Image))

{

FunctionSymbol funSymbol = (FunctionSymbol)this[functionName.Image];

if (paramTokens.Count() != funSymbol.Parameter.Count)

throw new UnMatchedParametersException(functionName, paramTokens);

for (int i = 0; i < paramSymbols.Count; i++)

{

if (paramSymbols[i].Type != funSymbol.Parameter[i].Type)

throw new UnMatchedParametersException(functionName, paramTokens[i]);

}

return funSymbol;

}

else

throw new UnDeclaredTypeException(functionName);

}

public Symbol GetMembeSymbol(string structTypeName, GrammerTreeNode memberNode, out int offset) {

if (this.ContainsKey(structTypeName))

{

StructSymbol symbol =(StructSymbol) this[structTypeName];

Symbol memberSymbol = null;

if ((memberSymbol = symbol.Children.Find(c => c.Name == memberNode.TerminalType.Image && c.Type == SemanticAnalyzer.CastNodeToSymbolType(memberNode))) == null)

throw new UnMatchedMemberException(memberNode.TerminalType, symbol);

else

{

offset = symbol.Children.IndexOf(memberSymbol);

return memberSymbol;

}

}

else if (this.m\_supTable != null)

{

return this.m\_supTable.GetMembeSymbol(structTypeName, memberNode, out offset);

}

else

throw new Exception(string.Concat("找不到定义为", structTypeName, "的类型,异常在FindDefinition中发生"));

}

public FunctionSymbol GetEntranceFncSymbol()

{

if (this.Entrance == null)

if (this.m\_supTable == null)

throw new SemanticException("缺少Main函数, 没有找到代码入口");

else

return this.m\_supTable.GetEntranceFncSymbol();

else

return this.Entrance;

}

### 3.1.7 在符号表中查找元素

public bool Contains(string name) {

if (this.ContainsKey(name))

return true;

else if (this.m\_supTable != null)

return this.m\_supTable.Contains(name);

else return false;

}

public Symbol Find(Token token) {

if (this.ContainsKey(token.Image))

return this[token.Image];

else if (this.m\_supTable != null)

return this.m\_supTable.Find(token);

else

throw new UnDeclaredTypeException(token);

}

### 3.1.8 用于计算表达式值的两个函数

private List<Quaternion> generateExpression(GrammerTreeNode exprRootNode,ref string nextAddrAvailable,out Symbol outValue,SymbolTable table=null) {

List<Quaternion> middleCode=new List<Quaternion>();

middleCode.Add(new Quaternion(Quaternion.DEBUG\_generateExpression));

table = table ?? this.m\_rootTable;

NonterminalType[] nonRecursiveType = { NonterminalType.FUNCTION\_CALL\_STMT, NonterminalType.STRUCT, NonterminalType.ARRAY };

if (!nonRecursiveType.Contains(exprRootNode.NonterminalType) && exprRootNode.Type!= TreeNodeType.TERMINAL)

{

middleCode.Add(new Quaternion(Quaternion.MALLOC, Quaternion.OBJ, Quaternion.STACK));//allocate a place to hold temp return value

outValue = new Symbol() { Address = Quaternion.AddrIncrease(ref nextAddrAvailable) };

if (exprRootNode.Children.Count == 2)

{

GrammerTreeNode operatorNode = exprRootNode.Children[0];

GrammerTreeNode operandNode = exprRootNode.Children[1];

string Operator = Quaternion.GetUnaryOperatorByOperatorNode(operatorNode);

Symbol Operand;

if (nonRecursiveType.Contains(operandNode.NonterminalType) || operandNode.Type == TreeNodeType.TERMINAL)

middleCode.AddRange(generateCalcElementAddress(operandNode, ref nextAddrAvailable, out Operand, table));

else

middleCode.AddRange(generateExpression(exprRootNode, ref nextAddrAvailable, out Operand, table));

middleCode.AddRange(generateTypeCheckAndConvert(Operator, Operand, ref outValue));

}

else if (exprRootNode.Children.Count == 3)

{

GrammerTreeNode operand1 = exprRootNode.Children[0];

GrammerTreeNode operatorNode = exprRootNode.Children[1];

GrammerTreeNode operand2 = exprRootNode.Children[2];

string Operator = Quaternion.GetBinoOperatorByOperatorNode(operatorNode);

Symbol Operand1;

Symbol Operand2;

if (nonRecursiveType.Contains(operand1.NonterminalType) || operand1.Type == TreeNodeType.TERMINAL)

middleCode.AddRange(generateCalcElementAddress(operand1, ref nextAddrAvailable, out Operand1, table));

else

middleCode.AddRange(generateExpression(operand1, ref nextAddrAvailable, out Operand1, table));

if (nonRecursiveType.Contains(operand2.NonterminalType) || operand2.Type == TreeNodeType.TERMINAL)

middleCode.AddRange(generateCalcElementAddress(operand2, ref nextAddrAvailable, out Operand2, table));

else

middleCode.AddRange(generateExpression(operand2, ref nextAddrAvailable, out Operand2, table));

middleCode.AddRange(generateTypeCheckAndConvert(Operator, Operand1, Operand2, ref outValue));

}

}

else

{

middleCode.AddRange(generateCalcElementAddress(exprRootNode, ref nextAddrAvailable, out outValue, table));

}

return middleCode;

}}

## 3.2 具体实现

### 3.2.1 分析入口

public string codeGen() {

emitComment("\*This is the interminal code:\r\n");

emitComment("Standard prelude:\r\n");

emitRM("LD", mp, 0, ac, "load maxaddress from location 0");

emitRM("ST", ac, 0, ac, "clear location 0");

emitComment("End of standard prelude.");

emitSkip(1);

cGen();//invoke the driver

int crnLoc = emitSkip(0);

emitBackup(rtnInsLoc);

emitRM("LDC",ac,crnLoc,0,"Load the return address");

emitBackup(rtnInsLoc+1);

emitRM("ST",ac,1,gp,"store the return address");

emitRestore();

emitRO("HALT",0,0,0,"end of the program");

return this.codeString;

}

### 3.2.2 程序由类和结构体构成

public void genClass(GrammerTreeNode node, string funName)

{

foreach (GrammerTreeNode subNode in node.Children) {

if ((subNode.Type == TreeNodeType.NONTERMINAL) && (subNode.NonterminalType == NonterminalType.FUNCTION\_DEFINE\_STMT)) {

genFunctionDefineStmt(subNode, funName);

}

}

}

public void genStruct(GrammerTreeNode node, string funName)

{

//we set the start location of struct is -1

globalTable.insertStruct(node,-1);

}

public void genFunctionDefineStmt(GrammerTreeNode node, string funName)

{

emitComment("->Function Definition: " + node.Children[1].TerminalType.Image);

globalTable.insertFunction(node,emitLoc);

//build a symbol table for this function

symbolTable = new SymTab(globalTable);

tableList.Add(symbolTable);//add it to the table list

//reset the location

location = 0;

//for the start switch

if (node.Children[1].TerminalType.Image.Equals("main"))

{

int currentLoc = emitSkip(0);

emitBackup(2);

emitRM\_Abs("LDA", pc, currentLoc, "jmp to main");

emitRestore();

//push the gp to stack

emitRM("LDC", ac, 0, 0, "Load constant 0");

emitRM("ST", ac, 0, gp, "store to stack");

location++;

rtnInsLoc = emitSkip(2);//skip for return and store it

location++;

}

else {

location += 2;//increment 2

BucketList temL = globalTable.lookUp(node.Children[1].TerminalType.Image);

location += ((List<Hashtable>)temL.hashTable["params"]).Count;

}

if((node.Children.Last().Type == TreeNodeType.NONTERMINAL)&&(node.Children.Last().NonterminalType == NonterminalType.STMT\_SEQUENCE)){

genStmtSequence(node.Children.Last(), node.Children[1].TerminalType.Image);//generate the stmt sequence

}

emitComment("<-Function Definition: " + node.Children[1].TerminalType.Image + "\r\n");

}}

### 3.2.3 单条语句

public void genStmt(GrammerTreeNode node,string funName) {

if (node.Type == TreeNodeType.NONTERMINAL) {

switch (node.NonterminalType) {

case NonterminalType.CLASS:

genClass(node,funName);

break;

case NonterminalType.STRUCT:

genStruct(node, funName);

break;

case NonterminalType.FUNCTION\_DEFINE\_STMT:

genFunctionDefineStmt(node, funName);

break;

case NonterminalType.STMT\_SEQUENCE:

genStmtSequence(node, funName);

break;

case NonterminalType.STMT:

genStmt(node.Children[0], funName);

break;

case NonterminalType.IF\_STMT:

genIfStmt(node, funName);

break;

case NonterminalType.WHILE\_STMT:

genWhileStmt(node, funName);

break;

case NonterminalType.DECLARE\_STMT:

genDeclareStmt(node, funName);

break;

case NonterminalType.READ\_STMT:

genReadStmt(node, funName);

break;

case NonterminalType.WRITE\_STMT:

genWriteStmt(node, funName);

break;

case NonterminalType.FUNCTION\_CALL\_STMT:

genFunctionCallStmt(node, funName);

break;

case NonterminalType.RETURN\_STMT:

genReturnStmt(node, funName);

break;

case NonterminalType.ASSGIN\_STMT:

genAssignStmt(node, funName);

break;

case NonterminalType.STRUCT\_DEC\_STMT:

genStructDecStmt(node, funName);

break;

case NonterminalType.STRUCT\_ASSIGN\_STMT:

genStructAssignStmt(node, funName);

break;

}

}

}}

### 3.2.4 语句序列

public void genStmtSequence(GrammerTreeNode node, string funName)

{

if (node != null)

{

foreach (GrammerTreeNode subNode in node.Children)

genStmt(subNode, funName);

}

}}

### 3.2.5 声明语句

public void genDeclareStmt(GrammerTreeNode node,string funName)

{

//update the symbol table

symbolTable.insertSymbol(node,location);

if (node.Children.Count > 2) {

//assign

genExp(node.Children[2], funName);

emitRM("ST", ac, location, gp, "declare assign");

}

//update the value of the location

location += symbolTable.lookUpSize(node.Children[1].TerminalType.Image);

}

}

### 3.2.6 表达式

public void genExp(GrammerTreeNode node, string funName)

{

if (node.Type == TreeNodeType.NONTERMINAL)

{

switch (node.NonterminalType) {

case NonterminalType.OREXPR:

genOrExp(node, funName);

break;

case NonterminalType.ANDEXPR:

genAndExp(node, funName);

break;

case NonterminalType.LOGICNOTEXPR:

genLogicalNotExp(node, funName);

break;

case NonterminalType.EQUALEXPR:

genEqualExp(node, funName);

break;

case NonterminalType.ARITHMEXPR:

genArithmExp(node, funName);

break;

case NonterminalType.TERM:

genTermExp(node, funName);

break;

case NonterminalType.UNARY:

genUnaryExp(node, funName);

break;

case NonterminalType.STRUCT\_MEM\_DESC:

genStructMemDescExp(node, funName);

break;

case NonterminalType.ARRAY:

genIdentifier(node,funName);

break;

default:

genStmt(node,funName);//generate the other stmts

break;

}

}

else {

switch(node.TerminalType.Kind){

case TokenKind.TRUE:

genTrue(node, funName);

break;

case TokenKind.FALSE:

genFalse(node, funName);

break;

case TokenKind.INT\_CONSTANT:

genIntConstant(node, funName);

break;

case TokenKind.REAL\_CONSTANT:

genRealConstant(node, funName);

break;

case TokenKind.IDENTIFIER:

genIdentifier(node, funName);

break;

default:

break;

}

}

}

///the tool method for the last generator}

### 3.2.7 运算表达式

public void genArithmExp(GrammerTreeNode node, string funName)

{

//gen the left

emitComment("->Arithmatic");

genExp(node.Children[0], funName);

emitRM("ST", ac, tmpOffset--, mp, "store left opto");

genExp(node.Children[2], funName);

emitRM("LD", ac1, ++tmpOffset, mp, "op: load left opto");

if (node.Children[1].TerminalType.Kind == TokenKind.PLUS)

emitRO("ADD", ac, ac1, ac, "op +");

else

emitRO("SUB", ac, ac1, ac, "op -");

emitComment("<-Arithmatic");

}

}

### 3.2.8 分表达式

public void genTermExp(GrammerTreeNode node, string funName)

{

//gen the left

emitComment("->Term");

genExp(node.Children[0], funName);

emitRM("ST", ac, tmpOffset--, mp, "store left opto");

genExp(node.Children[2], funName);

emitRM("LD", ac1, ++tmpOffset, mp, "op: load left opto");

if (node.Children[1].TerminalType.Kind == TokenKind.MULTIPLY)

emitRO("MUL", ac, ac1, ac, "op \*");

else if (node.Children[1].TerminalType.Kind == TokenKind.DIVIDE)

emitRO("DIV",ac,ac1,ac, "op /");

else //for mod

emitRO("MOD", ac, ac1, ac, "op %");

emitComment("<-Term");

}

public void genUnaryExp(GrammerTreeNode node, string funName)

{

if (node.Children[0].TerminalType.Kind == TokenKind.MINUS) {

emitComment("->Negative value");

genExp(node.Children[1], funName);

emitRM("ST", ac, tmpOffset--,mp, "store pos value");

emitRM("LDC", ac, 0, 0, "load const 0");

emitRM("LD", ac1, ++tmpOffset,mp, "op: load pos value");

emitRO("SUB", ac, ac, ac1, "op -");

emitComment("<-Negative value");

}

}

}

### 3.2.9 返回语句

public void genReturnStmt(GrammerTreeNode node, string funName)

{

//emitRM("LD", ac1, 1, gp, "load the return address");

//emitRM("ST", ac1, tmpOffset--, mp, "store the return address");

if(node.Children.Count > 0)

genExp(node.Children[0],funName);

//emitRM("ST",ac1,tmpOffset--,mp,"store the return address");

emitRM("LD", ac1, 1, gp, "load the return address");

emitRM("LD",gp,0,gp,"back the gp");

//emitRM("LD",ac1,++tmpOffset,mp,"load return address");

//jmp to the return address

emitRM("LDA",pc,0,ac1,"jmp return");

}}

### 3.2.10 循环语句

public void genWhileStmt(GrammerTreeNode node, string funName)

{

int savedLoc1, savedLoc2, currentLoc;

int loc;

emitComment("-> While");

savedLoc1 = emitSkip(0);

genExp(node.Children[0],funName);//gen the exp

emitComment("while: jump to end belongs here");

savedLoc2 = emitSkip(1);

genStmtSequence(node.Children[1],funName);//gen the stmt sequence

emitRM\_Abs("LDA", pc, savedLoc1, "jmp to start");

//backup the while stmt

currentLoc = emitSkip(0);

emitBackup(savedLoc2);

emitRM\_Abs("JEQ", ac, currentLoc, "while: jmp to end");//jump to the end

emitRestore();

emitComment("<- While");

}

### 3.2.11 条件语句

public void genIfStmt(GrammerTreeNode node, string funName)

{

int savedLoc1, savedLoc2, currentLoc;

int loc;

emitComment("-> If");

genExp(node.Children[0], funName);//gen the exp

savedLoc1 = emitSkip(1);

emitComment("if: jump to else belongs here");

genStmtSequence(node.Children[1],funName);

savedLoc2 = emitSkip(1);

emitComment("if: jump to end belongs here");

//backup for if-stmt

currentLoc = emitSkip(0);

emitBackup(savedLoc1);

emitRM\_Abs("JEQ", ac, currentLoc, "if: jmp to else");

emitRestore();

if (node.Children.Count == 3) { //has elst part

genStmtSequence(node.Children[2],funName);

}

//backup for else part,to the end

currentLoc = emitSkip(0);

emitBackup(savedLoc2);

emitRM\_Abs("LDA", pc, currentLoc, "jmp to end");

emitRestore();

emitComment("<- If");

}

}

### 3.2.12 布尔表达式

public void genOrExp(GrammerTreeNode node, string funName)

{

emitComment("->OrExp");

genExp(node.Children[0], funName);

int savedLoc1 = emitSkip(1);

genExp(node.Children[2], funName);

int cntLoc = emitSkip(0);

emitBackup(savedLoc1);

emitRM\_Abs("JNE", ac, cntLoc, "andExp is true");

emitRestore();

emitRM("LDC", ac, 1, ac, "set ac false");

emitComment("<-OrExp");

}

public void genAndExp(GrammerTreeNode node, string funName)

{

emitComment("->AndExp");

genExp(node.Children[0],funName);

int savedLoc1 = emitSkip(1);

genExp(node.Children[2],funName);

int cntLoc = emitSkip(0);

emitBackup(savedLoc1);

emitRM\_Abs("JEQ",ac,cntLoc,"andExp is false");

emitRestore();

emitRM("LDC",ac,0,ac,"set ac false");

emitComment("<-AndExp");

}

public void genLogicalNotExp(GrammerTreeNode node, string funName)

{

emitComment("->NotExp");

genExp(node.Children[1],funName);

emitRM("JEQ",ac,2,pc,"not exp");

emitRM("LDC",ac,0,ac,"set false");

emitRM("LDA",pc,1,pc,"step over");

emitRM("LDC",ac,1,ac,"set true");

emitComment("<-NotExp");

}

public void genEqualExp(GrammerTreeNode node, string funName)

{

emitComment("->EqualExp");

genExp(node.Children[0],funName);

emitRM("ST",ac,tmpOffset--,mp,"store left optr");

genExp(node.Children[2],funName);

emitRM("LD",ac1,++tmpOffset,mp,"load left optr");

emitRO("SUB", ac,ac1,ac, "opr -");

if (node.Children[1].TerminalType.Kind == TokenKind.EQ) {

emitRM("JEQ", ac, 2, pc, "br if true");

}

else if (node.Children[1].TerminalType.Kind == TokenKind.NEQ)

{

emitRM("JNE", ac, 2, pc, "br if true");

}

else if (node.Children[1].TerminalType.Kind == TokenKind.GOE)

{

emitRM("JGE", ac, 2, pc, "br if true");

}

else if (node.Children[1].TerminalType.Kind == TokenKind.GT)

{

emitRM("JGT", ac, 2, pc, "br if true");

}

else if (node.Children[1].TerminalType.Kind == TokenKind.LOE)

{

emitRM("JLE", ac, 2, pc, "br if true");

}

else {

emitRM("JLT", ac, 2, pc, "br if true");

}

emitRM("LDC", ac, 0, ac, "false case");

emitRM("LDA", pc, 1, pc, "unconditional jmp");

emitRM("LDC", ac, 1, ac, "load constant true");

emitComment("<-EqualExp");

}

### 3.2.13 读语句

public void genReadStmt(GrammerTreeNode node,string funName)

{

emitRO("IN", ac, 0, 0, "read value");

int loc;

//#true

if ((node.Children[0].Type == TreeNodeType.NONTERMINAL) && (node.Children[0].NonterminalType == NonterminalType.STRUCT\_MEM\_DESC)){

//for the struct desc

GrammerTreeNode memNode = node.Children[0];

loc = symbolTable.interpretStructMemberLocation(memNode);

}

//#true

else if((node.Children[0].Type == TreeNodeType.NONTERMINAL)&&(node.Children[0].Children.Count

!= 0))

{ //id element follow

loc = symbolTable.interpretElementLocation(node.Children[0]);

}

//#true

else { //symbol

//find in the params

if ((loc = symbolTable.lookUpParaLoc(funName, node.Children[0].TerminalType.Image)) == -1)

{

loc = symbolTable.lookUpLoc(node.Children[0].TerminalType.Image);

}

else ;//如果是参数，则已经为相对地址。存入符号表的都是绝对地址。指令中的都为相对地址

}

emitRM("ST",ac, loc,gp, "read: store value");

}

}

### 3.2.14 写语句

public void genWriteStmt(GrammerTreeNode node, string funName)

{

if(node.Children.Count != 0)

genExp(node.Children[0], funName);

emitRO("OUT",ac, 0, 0, "write ac");

}

public void genFunctionCallStmt(GrammerTreeNode node, string funName)

{

if ((node.Type == TreeNodeType.NONTERMINAL) && (node.NonterminalType == NonterminalType.FUNCTION\_CALL\_STMT)) {

emitComment("->Function Call: "+node.Children[0].TerminalType.Image);

//push the parameters to temporary var

for (int i = 1; i < node.Children.Count; i++)

{

genExp(node.Children[i], funName);

emitRM("ST", ac, tmpOffset--, mp, "store parameter " + i + " to tmp");

}

int temLoc = 0;

//如何更新函数ebp的值，先存老的到tmp，然后更新之后存入位置0

emitRM("ST",gp,tmpOffset--,mp,"store the old gp to tmp");

//update the value of the ebp

emitRM("LDC",ac,location,0,"load the location");

emitRO("ADD",gp,ac,gp,"update the gp");

emitRM("LD",ac1,++tmpOffset,mp,"Load the old gp to ac1");

emitRM("ST",ac1,0,gp,"store the old gp to loc 0");

temLoc++;

int savedLoc1 = emitSkip(2);

temLoc++;

for (int i = 1; i < node.Children.Count; i++)

{

emitRM("LD", ac, ++tmpOffset, mp, "Load parameter " + (node.Children.Count-i));

emitRM("ST", ac, temLoc, gp, "store parameter " + (node.Children.Count - i));

temLoc++;

}

//go to the func instruction

int funStart = symbolTable.lookUpLoc(node.Children[0].TerminalType.Image);

emitRM\_Abs("LDA", pc, funStart, "jmp to function: " + node.Children[0].TerminalType.Image);

//push the return address

int currentLoc = emitSkip(0);

emitBackup(savedLoc1);

emitRM("LDC",ac,currentLoc,0,"load return address");

emitBackup(savedLoc1+1);

emitRM("ST",ac,1,gp,"store the return address");

emitRestore();

emitComment("<-Function Call: " + node.Children[0].TerminalType.Image);

}

}}

### 3.2.15 赋值语句

public void genAssignStmt(GrammerTreeNode node, string funName)

{

emitComment("-> assign");

int loc;

//calculate the value of the expression

genExp(node.Children[1], funName);

if(node.Children[1].NonterminalType != NonterminalType.FUNCTION\_CALL\_STMT){

//change the value stored in the table

symbolTable.change(node.Children[0].TerminalType.Image,symbolTable.interpreteDimenExpression(node.Children[1]));

}

if (node.Children[0].Children.Count != 0)

{ //array element

loc = symbolTable.interpretElementLocation(node.Children[0]);

emitRM("ST",ac, loc,gp, "assign: store value to array element");

}

else {

if ((loc = symbolTable.lookUpParaLoc(funName, node.Children[0].TerminalType.Image)) == -1)

{

loc = symbolTable.lookUpLoc(node.Children[0].TerminalType.Image);

}

emitRM("ST", ac, loc, gp, "assign: store value to symbol");

}

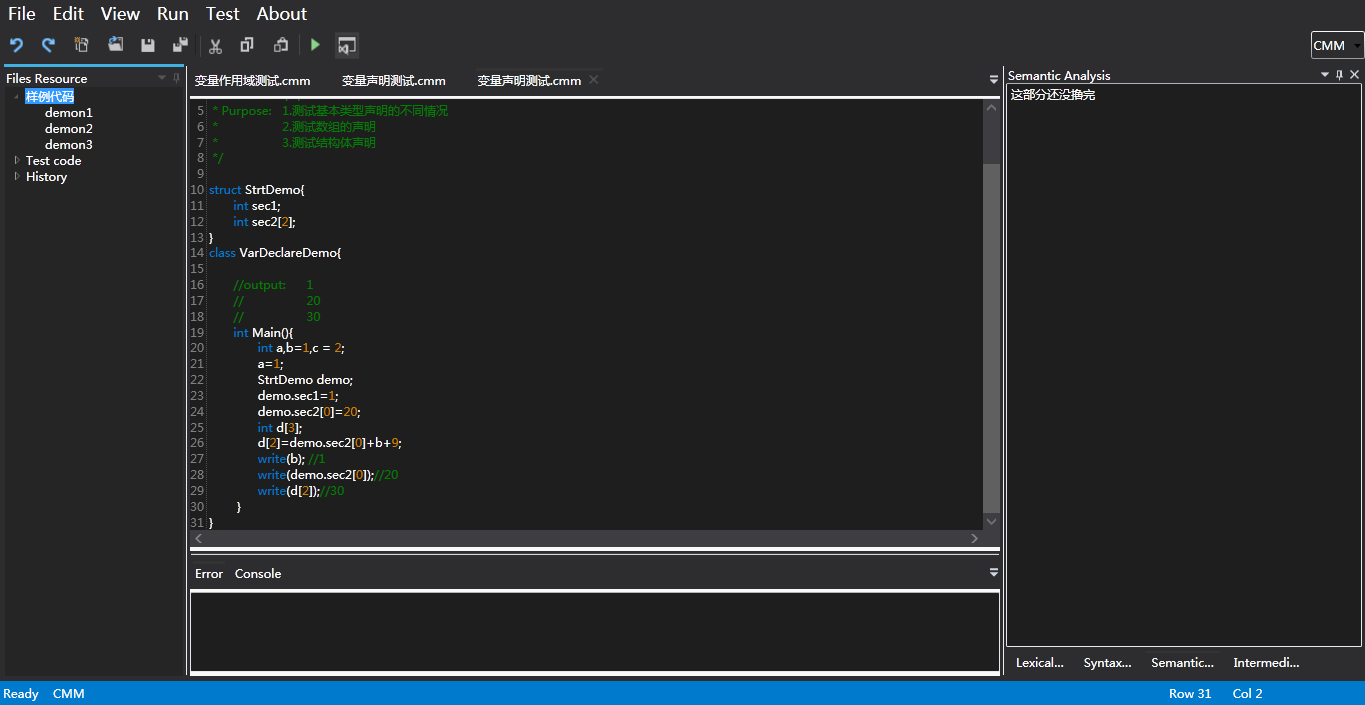
emitComment("<- assign");

}

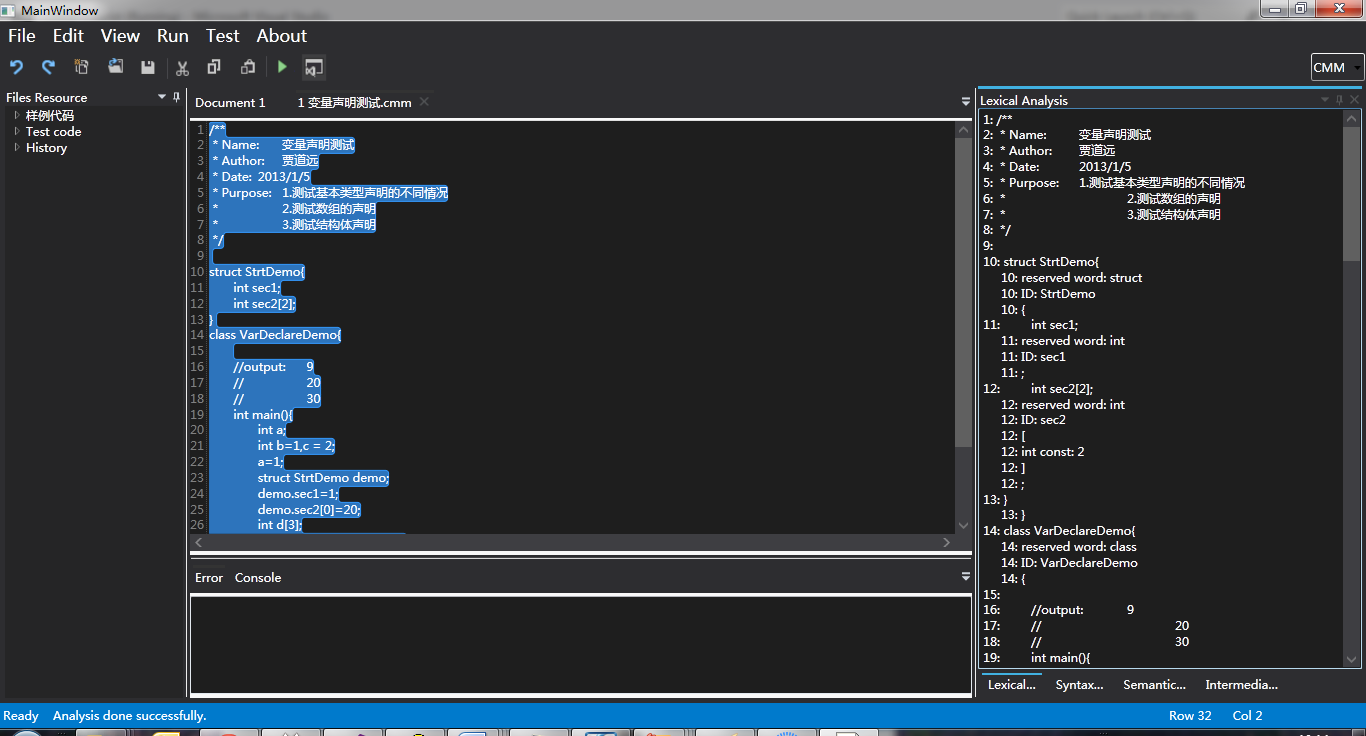
# 4 实例展示

## 4.1 无参数类展示

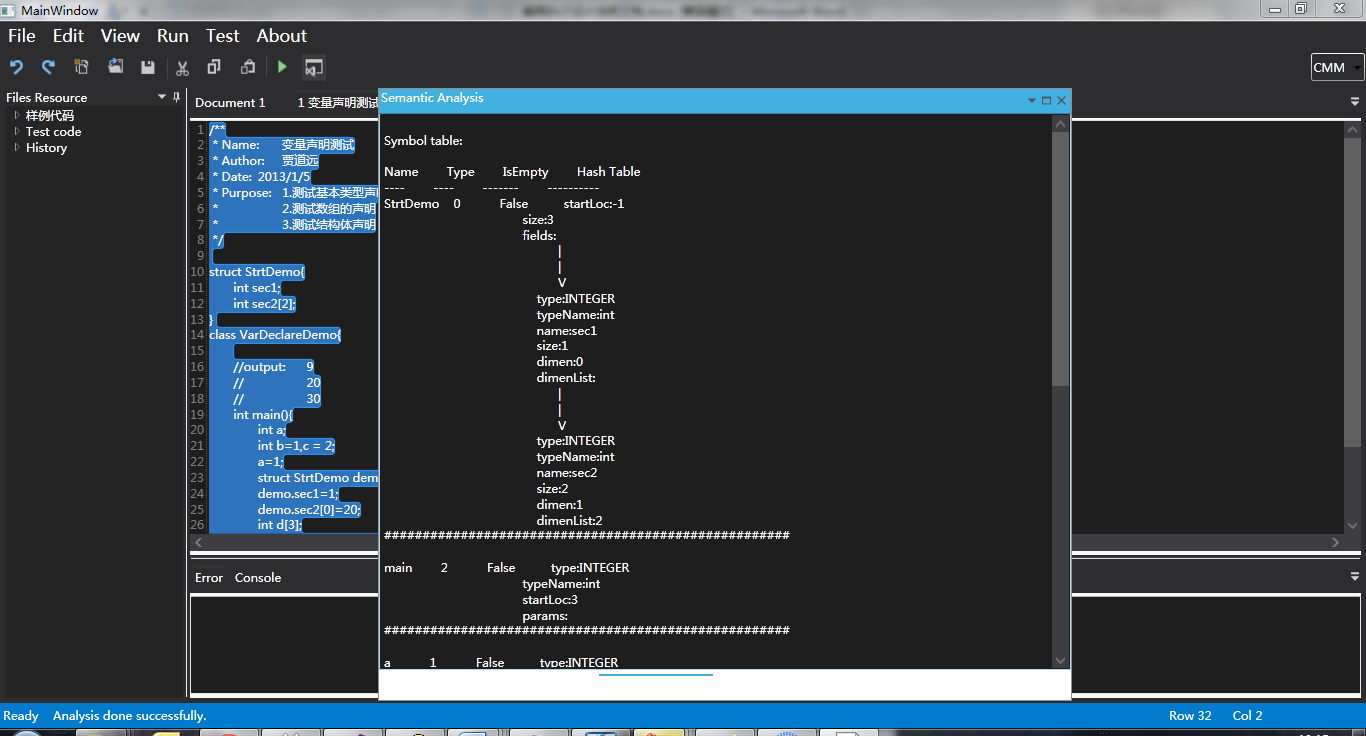
输入以下内容：



得到如下结果，完成分析过程：

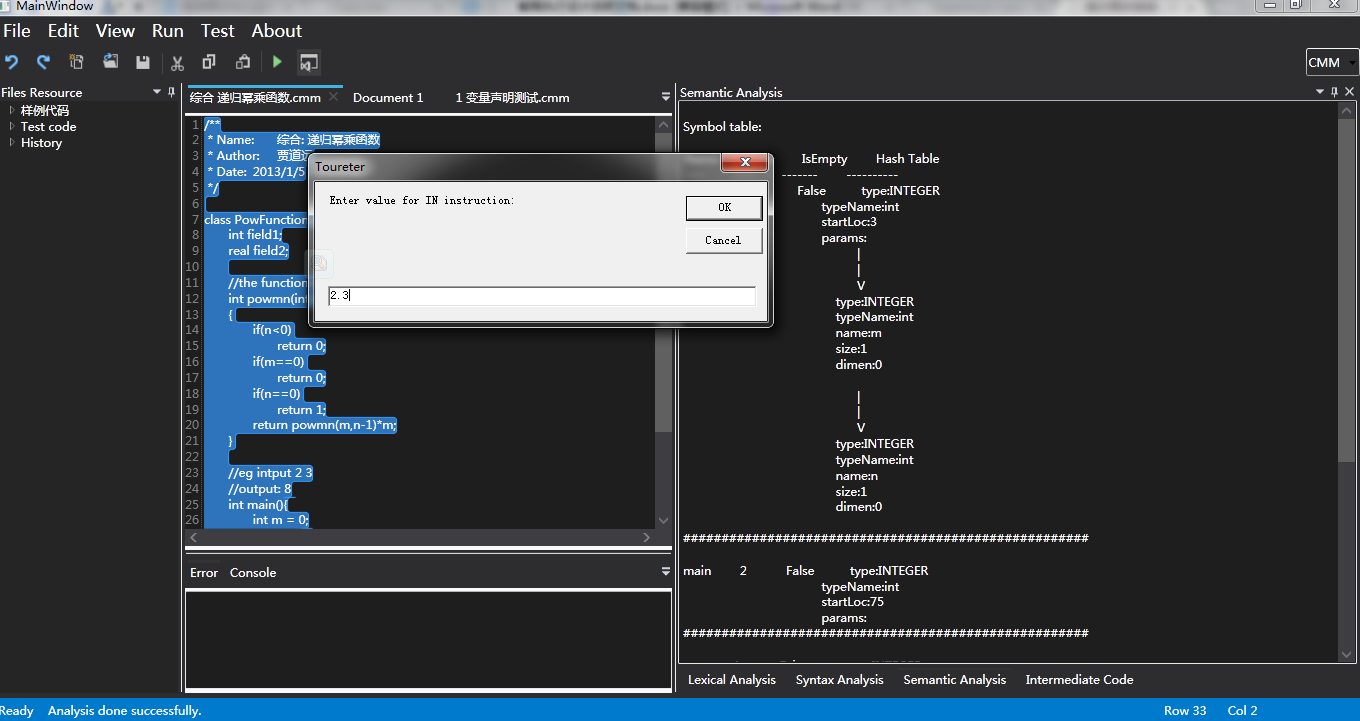


所得到的符号表是：

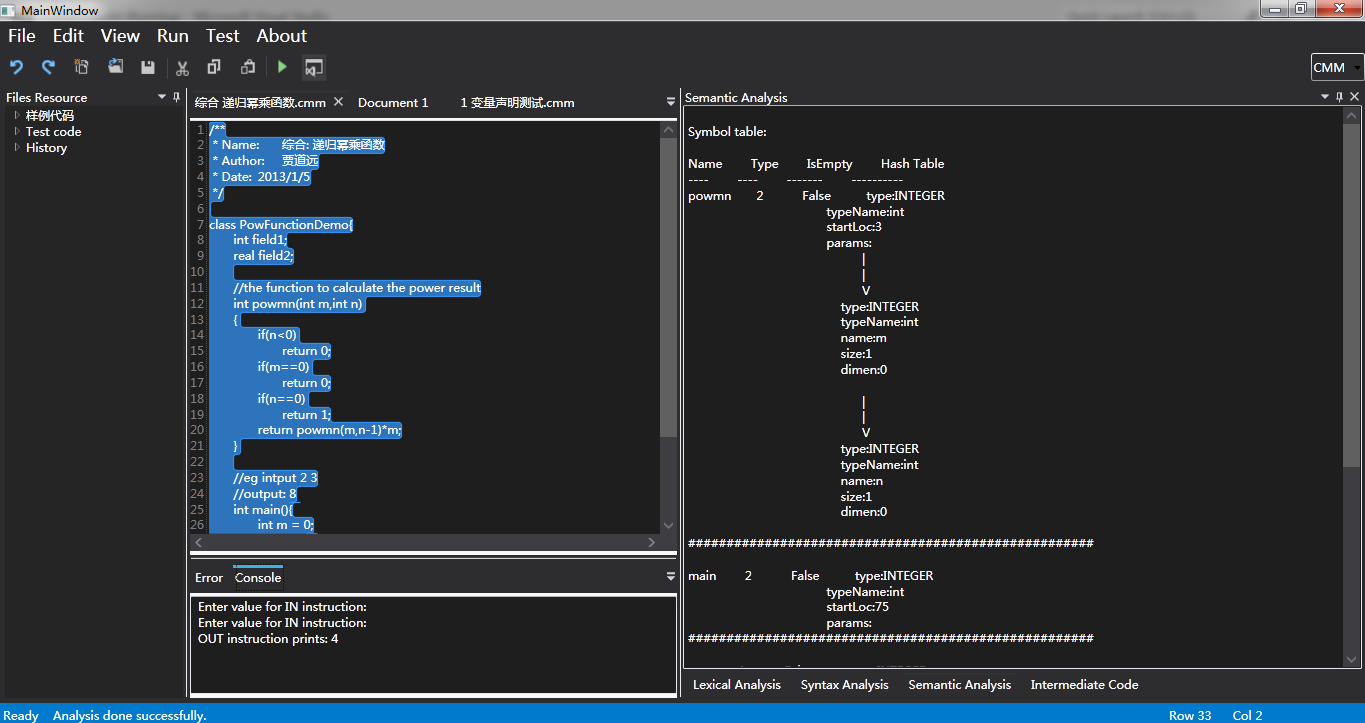


## 4.2 有参数类展示

输入以下内容，得到输入内容和参数输入截图如下：



执行结果如下图：



中间代码如下图：

