# 目录

[1 目录 1](#_Toc513585289)

[2 语义分析详细设计 4](#_Toc513585290)

[2.1 符号表设计 4](#_Toc513585291)

[2.1.1 主符号表分析 4](#_Toc513585292)

[2.1.2 子符号表分析 5](#_Toc513585293)

[2.1.3 符号表分析总结 5](#_Toc513585294)

[2.1.4 在符号表中存放当前程序名的必要性 5](#_Toc513585295)

[2.1.5 表项分析 6](#_Toc513585296)

[2.1.6 表类定义 6](#_Toc513585297)

[2.1.6.1 变量 6](#_Toc513585298)

[2.1.6.2 添加传值参数方法 7](#_Toc513585299)

[2.1.6.3 添加引用参数方法 7](#_Toc513585300)

[2.1.6.4 添加普通变量方法 8](#_Toc513585301)

[2.1.6.5 添加常量方法 8](#_Toc513585302)

[2.1.6.6 添加数组方法 9](#_Toc513585303)

[2.1.6.7 添加过程方法 10](#_Toc513585304)

[2.1.6.8 添加函数方法 10](#_Toc513585305)

[2.1.6.9 添加子程序符号表指针方法 11](#_Toc513585306)

[2.1.6.10 添加主/子程序名方法 11](#_Toc513585307)

[2.1.6.11 添加主程序参数方法 12](#_Toc513585308)

[2.1.7 表项类定义 13](#_Toc513585309)

[2.1.7.1 变量 13](#_Toc513585310)

[2.1.7.2 各种类标识符与其拥有的属性情况 13](#_Toc513585311)

[2.1.7.3 设置为传值参数方法 14](#_Toc513585312)

[2.1.7.4 设置为引用参数方法 15](#_Toc513585313)

[2.1.7.5 设置为普通变量方法 15](#_Toc513585314)

[2.1.7.6 设置为常量方法 16](#_Toc513585315)

[2.1.7.7 设置为数组方法 16](#_Toc513585316)

[2.1.7.8 设置为过程方法 17](#_Toc513585317)

[2.1.7.9 设置为函数方法 18](#_Toc513585318)

[2.1.7.10 设置为主/子程序名方法 18](#_Toc513585319)

[2.1.7.11 设置为主程序参数方法 19](#_Toc513585320)

[2.1.8 对外接口 20](#_Toc513585321)

[2.1.8.1 查找所在符号表位置 20](#_Toc513585322)

[2.1.8.2 检查是否与库程序名、主程序名、主程序参数同名 20](#_Toc513585323)

[2.1.8.3 找到第X个形式参数的类型 21](#_Toc513585324)

[2.1.8.4 检查第X个形式参数是否是引用调用 21](#_Toc513585325)

[2.1.8.5 检查第X维下标是否越界 21](#_Toc513585326)

[2.1.8.6 检查id是否是引用参数 22](#_Toc513585327)

[2.1.8.7 符号表指针 23](#_Toc513585328)

[2.1.9 符号表定位与重定位 23](#_Toc513585329)

[2.1.9.1 定位 23](#_Toc513585330)

[2.1.9.2 重定位 23](#_Toc513585331)

[2.2 库程序支持 23](#_Toc513585332)

[2.3 作用域检查 23](#_Toc513585333)

[2.3.1 作用域规则 23](#_Toc513585334)

[2.3.2 重定义检查 23](#_Toc513585335)

[2.3.2.1 库程序名、主程序名、主程序参数名的重定义检测 24](#_Toc513585336)

[2.3.2.2 子程序名 24](#_Toc513585337)

[2.3.2.3 子程序内部定义符号 24](#_Toc513585338)

[2.3.2.4 检查范围 24](#_Toc513585339)

[2.3.3 未定义检查 24](#_Toc513585340)

[2.4 语法成分检查 24](#_Toc513585341)

[2.4.1 \_VariantReference种类检查 24](#_Toc513585342)

[2.4.2 \_Statement是否是返回语句 25](#_Toc513585343)

[2.4.2.1 赋值语句 25](#_Toc513585344)

[2.4.2.2 过程调用 25](#_Toc513585345)

[2.5 数量一致性检查 25](#_Toc513585346)

[2.5.1 数组维数 25](#_Toc513585347)

[2.5.2 子程序调用参数个数 25](#_Toc513585348)

[2.5.2.1 用户自定义程序 25](#_Toc513585349)

[2.5.2.2 库程序 25](#_Toc513585350)

[2.5.3 符号种类错误检查 26](#_Toc513585351)

[2.6 类型检查 26](#_Toc513585352)

[2.6.1 表达式类型检查 26](#_Toc513585353)

[2.6.1.1 \_Expression节点扩展 26](#_Toc513585354)

[2.6.1.2 \_VariantReference节点扩展 26](#_Toc513585355)

[2.6.1.3 \_FunctionCall节点扩展 27](#_Toc513585356)

[2.6.1.4 \_Expression节点类型分析 27](#_Toc513585357)

[2.6.1.5 \_VariantReference节点类型分析 28](#_Toc513585358)

[2.6.2 语句的类型检查 28](#_Toc513585359)

[2.6.2.1 AST语句类扩展 28](#_Toc513585360)

[2.6.2.2 各种语句类型分析 28](#_Toc513585361)

[2.6.3 程序调用参数类型检查 29](#_Toc513585362)

[2.6.3.1 用户自定义程序的调用 29](#_Toc513585363)

[2.6.3.2 库程序的调用 29](#_Toc513585364)

[2.6.4 函数返回值类型检查 30](#_Toc513585365)

[2.6.5 数组下标表达式类型检查 30](#_Toc513585366)

[2.6.6 循环变量类型检查 30](#_Toc513585367)

[2.7 涉及常量的语义分析 30](#_Toc513585368)

[2.7.1 常量定义时的常数传播 30](#_Toc513585369)

[2.7.1.1 常数传播的目标 30](#_Toc513585370)

[2.7.1.2 涉及的数据结构说明 30](#_Toc513585371)

[2.7.1.3 常量传播方法 31](#_Toc513585372)

[2.7.2 计算整型常量表达式的值 31](#_Toc513585373)

[2.7.2.1 涉及的数据结构说明 31](#_Toc513585374)

[2.7.2.2 标记整型常量表达式和计算取值的方法 31](#_Toc513585375)

[2.7.3 除0错误的静态检查 32](#_Toc513585376)

[2.7.4 数组下标越界的静态检查 32](#_Toc513585377)

[2.7.5 数组定义时上下界大小关系 32](#_Toc513585378)

[2.8 函数返回值语句存在性检测（警告信息） 32](#_Toc513585379)

[2.8.1 算法分析 32](#_Toc513585380)

[2.8.2 数据结构 33](#_Toc513585381)

[2.8.3 警告信息格式设计 33](#_Toc513585382)

[2.8.4 引用外部的函数接口 33](#_Toc513585383)

[2.8.4.1 整型数字转化为字符串 33](#_Toc513585384)

[2.8.5 内部函数接口 33](#_Toc513585385)

[2.8.5.1 检查当前语句是否是“返回值完备”的 33](#_Toc513585386)

[2.8.5.2 检查函数定义的程序体语句列表是否是“返回值完备”的 36](#_Toc513585387)

[2.8.5.3 检查函数定义是否是“返回值完备”的 36](#_Toc513585388)

[2.9 错误检查和恢复策略详述 37](#_Toc513585389)

[2.9.1 恢复策略 37](#_Toc513585390)

[2.9.1.1 形参重定义 37](#_Toc513585391)

[2.9.1.2 主程序名重定义 37](#_Toc513585392)

[2.9.1.3 其它标识符的重定义 37](#_Toc513585393)

[2.9.1.4 类型检查 37](#_Toc513585394)

[2.9.1.5 数组定义时下界大于上界 38](#_Toc513585395)

[2.9.2 各抽象语法树的语义分析策略详述 38](#_Toc513585396)

[2.9.2.1 \_VariantReference 38](#_Toc513585397)

[2.9.2.2 \_Satatement 38](#_Toc513585398)

[2.9.2.3 \_FormalParameter 39](#_Toc513585399)

[2.9.2.4 \_FunctionDefinition 39](#_Toc513585400)

[2.9.2.5 \_Variant 39](#_Toc513585401)

[2.9.2.6 \_Constant 39](#_Toc513585402)

[2.9.2.7 \_SubProgram 40](#_Toc513585403)

[2.9.2.8 \_FunctionCall 40](#_Toc513585404)

[2.9.2.9 \_Expression 40](#_Toc513585405)

[2.9.2.10 \_Program 41](#_Toc513585406)

[2.10 警告信息处理 41](#_Toc513585407)

[2.10.1 存储警告信息的数据结构 41](#_Toc513585408)

[2.10.2 警告信息格式 41](#_Toc513585409)

[2.11 错误信息处理 42](#_Toc513585410)

[2.11.1 存储错误信息的数据结构 42](#_Toc513585411)

[2.11.2 添加错误信息函数 42](#_Toc513585412)

[2.11.2.1 重定义 42](#_Toc513585413)

[2.11.2.2 未定义 42](#_Toc513585414)

[2.11.2.3 标识符类型错误 43](#_Toc513585415)

[2.11.2.4 数组下标、参数个数不匹配 43](#_Toc513585416)

[2.11.2.5 标识符种类错误 44](#_Toc513585417)

[2.11.2.6 表达式类型错误 45](#_Toc513585418)

[2.11.2.7 赋值语句两边类型不匹配错误 45](#_Toc513585419)

[2.11.2.8 数组下标越界 46](#_Toc513585420)

[2.11.2.9 数组下界比上界大 46](#_Toc513585421)

[2.11.2.10 运算符两边的操作数类型不一致 47](#_Toc513585422)

[2.11.2.11 某个操作数类型错误 47](#_Toc513585423)

[2.11.2.12 read函数参数出现错误 48](#_Toc513585424)

[2.11.2.13 除0错误 48](#_Toc513585425)

[2.11.2.14 read读取boolean类型错误 49](#_Toc513585426)

[2.11.2.15 直接添加错误 49](#_Toc513585427)

[2.12 语义分析函数设计 51](#_Toc513585428)

[2.12.1 常量语义分析 51](#_Toc513585429)

[2.12.2 变量语义分析 52](#_Toc513585430)

[2.12.3 语句语义分析 53](#_Toc513585431)

[2.12.4 表达式语义分析 58](#_Toc513585432)

[2.12.5 程序语义分析 62](#_Toc513585433)

[2.12.6 分程序语义分析 63](#_Toc513585434)

[2.12.7 子程序语义分析 64](#_Toc513585435)

[2.12.8 函数调用语义分析 65](#_Toc513585436)

[2.12.9 形式参数语义分析 67](#_Toc513585437)

[2.12.10 变量引用语义分析 67](#_Toc513585438)

# 语义分析详细设计

## 符号表设计

### 主符号表分析

主符号表首先需要保存全局常量、变量，子函数/过程。其次，为了保证库程序名、主程序名、主程序参数名不被重定义，主符号表中还需保存库程序项、主程序名项、主程序参数项。

各成分的存储顺序如下图所示：



注：amount指主程序参数的个数

### 子符号表分析

子符号表中，除了需要保存局部常量、变量等，还需保存子程序参数（传值参数、引用参数），当然为了避免子程序中的标识符与当前子程序名同名，还需将子程序名保存到子符号表中；由于我们不支持函数/过程的嵌套定义，所以子符号表不包含子程序这一项。

各成分的存储顺序如下图所示：



### 符号表分析总结

符号表的第0个位置，存放的必然是该符号表对应的程序名，第1到第amount个位置，存放的必然是该符号表对应的程序所定义的amount个参数；另外主符号表还将read、write、writeln、exit存放在了第amount+1到第amount+4个的位置。主符号表和子符号表接下来的位置都先存放了所有的常量定义，再存放了所有的变量定义。最后主符号表还保存了所有的子程序定义，按照文法规定，子过程和子函数的存放位置没有必然的先后关系，可以混合存储。

### 在符号表中存放当前程序名的必要性

首先需要保证程序定义中的所有符号都不能和程序名同名，这一点就足以支撑将当前程序名加入当前符号表，且需放在一开始的位置；其次，赋值语句有一种很特殊的情况，就是将函数名作为左值，表达式作为右值的情况，这实际上是一条返回值语句，在进行类型检查时，需要查符号表获得左值的类型，那么如果已经将当前的函数名保存在了当前的子符号表中，将检查类型检查中需要特判的情况，便于程序的编写。

### 表项分析

综合上述分析，我们发现主符号表和子符号表的逻辑结构差别不大，所以表类和表项类均统一设计。各种不同的标识符分析如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 标识符种类 | 是否可能出现在主符号表中 | 是否可能出现在子符号表中 | 附加属性 |
| 变量 | √ | √ | 变量类型 |
| 常量 | √ | √ | 常量类型、常量值（需根据常量类型设置不同的域）、常量是否是负数 |
| 数组 | √ | √ | 数组元素类型、数组维数、数组各维上下界 |
| 传值参数 | × | √ | 类型 |
| 引用参数 | × | √ | 类型 |
| 过程 | √ | × | 参数个数、指向子符号表的指针 |
| 函数 | √ | × | 参数个数、返回值类型、指向子符号表的指针 |
| 主/子程序名 | √ | √ | 程序类型（函数还是过程）、程序参数个数、返回值类型（如果是函数，则该项有意义） |
| 主程序参数 | √ | × | 无（主程序参数不带类型） |

注：附加属性指的是除了标识符、行号之外的属性。

### 表类定义

#### 变量

|  |  |  |
| --- | --- | --- |
| **名字** | **类型** | **描述** |
| **tableType** | string | 主符号表标识为“main”,子符号表标识为“sub” |
| **recordList** | vector<\_SymbolRecord\*> | \_SymbolRecord类型的vector，\_SymbolRecord即表项类。 用于存储每一个标识符的信息，符号表记录类型的数据结构会在下文具体描述 |
| **idToLoc** | map<string,int> | 存储标识符id及其位置对应关系的map结构。每加入一个符号，都要将其id和其所在recordList中的下标存储到该map中，可以加快查询速度。 |

#### 添加传值参数方法

* 接口

void addPara(string id, int lineNumber, string type)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 传值参数的id |
| int lineNumber | 传值参数所在行号 |
| string type | 传值参数的类型 |

* 函数功能

将传值参数类型的标识符加到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addPara(string id, **int** lineNumber, string type){
2. 遍历参数列表,查看当前标识符是否在符号表中已有记录{
3. 若已有记录，在当前标识符id的前面添加存储次数个\_;
4. }
5. 新建符号表记录；
6. 调用setPara将传值参数的id、行号、传值参数类型写入记录中；
7. 将记录添加到当前符号表中；
8. 将记录的位置保存到idToLoc中；
9. 若当前标识符在符号表中已有记录，存储次数加1;
10. 若当前标识符在符号表中未有记录，存储次数为1;
11. }

#### 添加引用参数方法

* 接口

void addVarPara(string id, int lineNumber, string type)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 传引用参数的id |
| int lineNumber | 传引用参数所在行号 |
| string type | 传引用参数的类型 |

* 函数功能

将传引用参数类型的标识符加到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addVarPara(string id, **int** lineNumber, string type){
2. 遍历参数列表,查看当前标识符是否在符号表中已有记录{
3. 若已有记录，在当前标识符id的前面添加存储次数个\_;
4. }
5. 新建符号表记录；
6. 调用setVarPara将传引用参数的id、行号、传引用参数类型写入记录中；
7. 将记录添加到当前符号表中；
8. 将记录的位置保存到idToLoc中；
9. 若当前标识符在符号表中已有记录，存储次数加1;
10. 若当前标识符在符号表中未有记录，存储次数为1;
12. }

#### 添加普通变量方法

* 接口

void addVar (string id, int lineNumber, string type)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 变量的id |
| int lineNumber | 变量所在行号 |
| string type | 变量的类型 |

* 函数功能

将变量类型的标识符加到符号表中

1. **void** \_SymbolTable::addVar (string id, **int** lineNumber, string type)
2. {
3. 新建符号表记录；
4. 调用setVar将变量的id、行号、变量类型写入记录中；
5. 将记录添加到当前符号表中；
6. 将记录的位置保存到idToLoc中；
7. }

#### 添加常量方法

* 接口

void addConst(string id, int lineNumber, string type, bool isMinusShow, string value)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 常量的id |
| int lineNumber | 常量所在行号 |
| string type | 常量的类型 |
| bool isMinusShow | 用于判断常量标识符符号正负的bool类型量 |
| string value | 常量标识符的值 |

* 函数功能

将常量类型的标识符加到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addConst(string id, **int** lineNumber, string type, **bool** isMinusShow, string value )
2. {           新建符号表记录；
3. 调用setConst将常量的id、行号、常量类型、判断正负号的**bool**量、
4. 常量的值写入记录中；
5. 将记录添加到当前符号表中；
6. 将记录的位置保存到idToLoc中；
7. }

#### 添加数组方法

* 接口

void addArray (string id, int lineNumber, string type, int amount, vector< pair<int, int> > arrayRangeList)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 数组的id |
| int lineNumber | 数组所在行号 |
| string type | 数组的类型 |
| int amount | 数组的维数 |
| vector< pair<int, int> > arrayRangeList | 存储数组各维上下界的vector |

* 函数功能

将数组类型的标识符加到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addArray(string id, **int** lineNumber, string type, **int** amount, vector< pair<**int**, **int**> > arrayRangeList)
2. {
3. 新建符号表记录；
4. 调用setArray将数组的id、行号、数组元素类型、数组维数、数组各维上下界写入记录中；
5. 将记录添加到当前符号表中；
6. 将记录的位置保存到idToLoc中；
7. }

#### 添加过程方法

* 接口

void addProcedure(string id, int lineNumber, int amount, \_SymbolTable \*subSymbolTable)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | Procedure的id |
| int lineNumber | Procedure所在行号 |
| int amount | Procedure参数的个数 |
| \_SymbolTable \*subSymbolTable=NULL | Procedure子符号表的指针 |

* 函数功能

将Procedure类型的标识符加到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addProcedure(string id, **int** lineNumber, **int** amount, \_SymbolTable \*subSymbolTable)
2. {
3. 新建符号表记录；
4. 调用setProcedure将过程的id、行号、过程参数个数、指向该过程子符号表的指针写入记录中；
5. 将记录添加到当前符号表中；
6. 将记录的位置保存到idToLoc中；
7. }

#### 添加函数方法

* 接口

void addFunction(string id, int lineNumber, string type, int amount, \_SymbolTable \*subSymbolTable)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | Function的id |
| int lineNumber | Function所在行号 |
| string type | Function返回值的类型 |
| int amount | Function参数的个数 |
| \_SymbolTable \*subSymbolTable=NULL | Function子符号表的指针 |

* 函数功能

将Function类型的标识符加到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addFunction(string id, **int** lineNumber, string type, **int** amount, \_SymbolTable \*subSymbolTable=NULL)
2. {
3. 新建符号表记录；
4. 调用setFunction将函数的id、行号、函数返回值类型、函数参数个数、
5. 指向该函数子符号表的指针写入记录中；
6. 将记录添加到当前符号表中；
7. 将记录的位置保存到idToLoc中；
8. }

#### 添加子程序符号表指针方法

* 接口

addSubSymbolTable(string id, \_SymbolTable \*subSymbolTable)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 子符号表的指针的id |
| \_SymbolTable \*subSymbolTable=NULL | 指向子符号表的指针 |

* 函数功能

将指向子符号表的指针加到主符号表中

* 函数伪代码

1. **void** \_SymbolTable::addSubSymbolTable(string id, \_SymbolTable\* symbolTable)
2. {
3. 判断idToloc中id是否已经存在，
4. 若存在，则将idToLoc[id]相应内容作为recordList的下标，将指向子符号表的指针放到符号表中；
5. 若不存在，则报错：子程序不存在；
6. }

#### 添加主/子程序名方法

* 接口

addProgramName(string id, int lineNumber, string subprogramType, int amount, string returnType)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 程序名的id |
| int lineNumber | 程序名所在行号 |
| string subprogramType | 该program的类型，只有Function和Procedure两种 |
| int amount | 该program的参数个数 |
| string returnType | 该program的返回值类型，若为Procedure，其返回值类型为void |

* 函数功能

将当前program的程序名添加到其符号表第0个位置上

* 函数伪代码

1. **void** \_SymbolTable::addProgramName(string id, **int** lineNumber, string subprogramType, **int** amount, string returnType) {
2. 判断记录列表是否为空，若不为空，则报错；
3. 遍历参数列表,查看当前标识符是否在符号表中已有记录{
4. 若已有记录，在当前标识符id的前面添加存储次数个\_;
5. }
6. 新建符号表记录；
7. 调用setProgramName将当前Program的id、行号、Program的类型（函数还是过程）、Program参数个数、Program返回值类型写入记录中；
8. 将记录添加到当前符号表中；
9. 将记录的位置保存到idToLoc中；
10. 若当前标识符在符号表中已有记录，存储次数加1;
11. 若当前标识符在符号表中未有记录，存储次数为1;
12. }

#### 添加主程序参数方法

* 接口

addVoidPara(string id, int lineNumber)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 主程序参数的id |
| int lineNumber | 主程序参数所在行号 |

* 函数功能

将主程序参数加入到符号表中

* 函数伪代码

1. **void** \_SymbolTable::addVoidPara(string id, **int** lineNumber)
2. {
3. 遍历参数列表,查看当前标识符是否在符号表中已有记录{
4. 若已有记录，在当前标识符id的前面添加存储次数个\_;
5. }
6. 新建符号表记录；
7. 调用setVoidPara将主函数参数的id、行号写入记录中；
8. 将记录添加到当前符号表中；
9. 将记录的位置保存到idToLoc中；
10. 若当前标识符在符号表中已有记录，存储次数加1;
11. 若当前标识符在符号表中未有记录，存储次数为1;
12. }

### 表项类定义

#### 变量

|  |  |  |
| --- | --- | --- |
| **名字** | **类型** | **描述** |
| flag | string | "value parameter"表示传值参数,  "var parameter"表示传引用参数,  "normal variant"表示普通变量,  "constant"表示常量,  "array"表示数组,  "procedure"表示过程,  "function"表示函数  "(sub)program name"表示该条记录是当前符号表对应的程序名信息  "parameter of program"表示主程序的参数 |
| id | string | 当前标识符的id |
| lineNumber | int | 当前标识符定义位置的行号 |
| type | string | 如果是变量/常量，则表示变量/常量类型；  如果是数组，则表示数组元素的类型；  如果是函数，则表示函数返回值类型，类型只能为基本类型,"integer","real","char","boolean" |
| value | string | 如果当前标识符是常量，则表示常量取值，这里面存的一定是无符号数（也就是非负数） |
| isMinusShow | bool | 表示常量前是否带负号，换句话说表示常量是否是负数 |
| amount | int | 如果是数组，则表示数组维数，如果是函数/过程，则表示参数个数 |
| arrayRangeList | vector< pair<int,int> > | 数组各维上下界 |
| subSymbolTable | \_SymbolTable\* | 指向过程/函数对应的子符号表的指针 |
| subprogramType | string | 这是一条特殊的记录，表示当前符号表对应的程序名称等信息，该变量表示程序是函数还是过程 |

#### 各种类标识符与其拥有的属性情况

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| flag | id | lineNumber | type | value | isMinusShow |
| value parameter | **√** | **√** | **√** | × | × |
| var parameter | **√** | **√** | **√** | × | × |
| normal variant | **√** | **√** | **√** | × | × |
| constant | **√** | **√** | **√** | √ | √ |
| array | **√** | **√** | **√** | × | × |
| procedure | **√** | **√** | **×** | × | × |
| fucntion | **√** | **√** | **√** | × | × |
| (sub)program name | **√** | **√** | **×** | × | × |
| parameter of program | √ | √ | × | × | × |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| flag | amount | arrayRangeList | subSymbolTable | subprogramType |
| value parameter | × | × | × | × |
| var parameter | × | × | × | × |
| normal variant | × | × | × | × |
| constant | × | × | × | × |
| array | √ | √ | × | × |
| procedure | √ | × | √ | × |
| fucntion | √ | × | √ | × |
| (sub)program name | √ | × | × | √ |
| parameter of program | × | × | × | × |

#### 设置为传值参数方法

* 接口

void setPara(string id, int lineNumber, string type)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 传值参数的id |
| int lineNumber | 传值参数所在行号 |
| string type | 传值参数的类型 |

* 函数功能

设置传值参数类型标识符所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setPara(string id, **int** lineNumber, string type) {
2. 将当前记录的flag设为"value parameter";
3. 将传入的传值参数id赋值给将当前记录的id;
4. 将传入的行号赋值给将当前记录的lineNumber；
5. 将传入的传值参数类型赋值给将当前记录的type；
6. }

#### 设置为引用参数方法

* 接口

void setVarPara(string id, int lineNumber, string type)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 传引用参数的id |
| int lineNumber | 传引用参数所在行号 |
| string type | 传引用参数的类型 |

* 函数功能
* 设置传引用参数类型标识符所在记录的具体信息
* 函数伪代码

1. **void** \_SymbolRecord::setPara(string id, **int** lineNumber, string type)
2. {
3. 将当前记录的flag设为" var parameter"；
4. 将传入的传引用参数id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的传引用参数类型赋值给将当前记录的type；
7. }

#### 设置为普通变量方法

* 接口

void setVar (string id, int lineNumber, string type)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 变量的id |
| int lineNumber | 变量所在行号 |
| string type | 变量的类型 |

* 函数功能

设置变量类型标识符所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setVar(string id, **int** lineNumber, string type)
2. {
3. 将当前记录的flag设为" normal variant"；
4. 将传入的变量id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的变量类型赋值给将当前记录的type；
7. }

#### 设置为常量方法

* 接口

void setConst(string id, int lineNumber, string type, bool isMinusShow, string value)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 常量的id |
| int lineNumber | 常量所在行号 |
| string type | 常量的类型 |
| bool isMinusShow | 用于判断常量标识符符号正负的bool类型量 |
| string value | 常量标识符的值 |

* 函数功能

设置常量类型标识符所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setConst(string id, **int** lineNumber, string type, **bool** isMinusShow, string value)
2. {
3. 将当前记录的flag设为"constant"；
4. 将传入的常量id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的常量类型赋值给将当前记录的type；
7. 将传入的记录常量正负号的isMinusShow赋值给当前记录的isMinusShow；
8. 将传入的常量的值赋值给当前记录的值；
9. }

#### 设置为数组方法

* 接口

void setArray (string id, int lineNumber, string type, int amount, vector< pair<int, int> > arrayRangeList)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 数组的id |
| int lineNumber | 数组所在行号 |
| string type | 数组的类型 |
| int amount | 数组的维数 |
| vector< pair<int, int> > arrayRangeList | 存储数组各维上下界的vector |

* 函数功能

设置数组类型标识符所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setArray(string id, **int** lineNumber, string type, **int** amount, vector< pair<**int**,**int**> > arrayRangeList)
2. {
3. 将当前记录的flag设为"array"；
4. 将传入的数组id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的数组元素类型赋值给将当前记录的type；
7. 将传入的数组维数赋值给当前记录的amount；
8. 将传入的数组记录各维上下界的值的vector赋值给当前记录的arrayRangeList；
9. }

#### 设置为过程方法

* 接口

void setProcedure(string id, int lineNumber, int amount, \_SymbolTable \*subSymbolTable=NULL)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | Procedure的id |
| int lineNumber | Procedure所在行号 |
| int amount | Procedure参数的个数 |
| \_SymbolTable \*subSymbolTable=NULL | Procedure子符号表的指针 |

* 函数功能

设置Procedure类型标识符所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setProcedure(string id, **int** lineNumber, **int** amount, \_SymbolTable \*subSymbolTable)
2. {
3. 将当前记录的flag设为"procedure"；
4. 将传入的过程id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的过程参数个数赋值给当前记录的amount；
7. 将传入的指向该过程子符号表指针赋值给当前记录的subSymbolTable；
8. }

#### 设置为函数方法

* 接口

void setFunction(string id, int lineNumber, string type, int amount, \_SymbolTable \*subSymbolTable=NULL)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | Function的id |
| int lineNumber | Function所在行号 |
| string type | Function返回值的类型 |
| int amount | Function参数的个数 |
| \_SymbolTable \*subSymbolTable=NULL | Function子符号表的指针 |

* 函数功能

设置Function类型标识符所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setFunction(string id, **int** lineNumber, string type, **int** amount, \_SymbolTable \*subSymbolTable)
2. {
3. 将当前记录的flag设为"function"；
4. 将传入的函数id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的函数返回值类型赋值给将当前记录的type；
7. 将传入的函数参数个数赋值给当前记录的amount；
8. 将传入的指向该函数子符号表指针赋值给当前记录的subSymbolTable；
9. }

#### 设置为主/子程序名方法

* 接口

setProgramName(string id, int lineNumber, string subprogramType, int amount, string returnType)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | program的id |
| int lineNumber | program所在行号 |
| string subprogramType | 该program的类型，只有Function和Procedure两种 |
| int amount | 该program的参数个数 |
| string returnType | 该program的返回值类型，若为Procedure，其返回值类型为void |

* 函数功能

设置当前program的程序名所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setProgramName(string id, **int** lineNumber, string subprogramType, **int** amount, string returnType)
2. {
3. 将当前记录的flag设为"(sub)program name"；
4. 将传入的主程序名id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. 将传入的程序类型赋值给将当前记录的type；
7. 将传入的程序参数个数赋值给当前记录的amount；
8. 将传入的该程序返回值的类型赋值给当前记录的returnType；
9. }

#### 设置为主程序参数方法

* 接口

setVoidPara(string id, int lineNumber)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 主程序参数的id |
| int lineNumber | 主程序参数所在行号 |

* 函数功能

设置主程序参数所在记录的具体信息

* 函数伪代码

1. **void** \_SymbolRecord::setVoidPara(string id, **int** lineNumber)
2. {
3. 将当前记录的flag设为"parameter of program"；
4. 将传入的主程序参数id赋值给将当前记录的id；
5. 将传入的行号赋值给将当前记录的lineNumber；
6. }

### 对外接口

#### 查找所在符号表位置

* 接口

\_SymbolRecord\* findSymbolRecord(\_SymbolTable\* currentSymbolTable, string id, int mode)

* 返回值

\_SymbolRecord\* findSymbolRecord

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_SymbolTable\* currentSymbolTable | 指向当前符号表的指针 |
| string id | 标识符id |
| int mode | 判断当前符号表是否为主符号表 |

* 函数功能

找出标识符在符号表中的位置

* 函数伪代码

#### 检查是否与库程序名、主程序名、主程序参数同名

* 接口

bool checkIsTheSameAsKey(string id, int lineNumber)

* 返回值

bool

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 标识符id |
| int lineNumber | 标识符行号 |

* 函数功能

检查标识符是否与库程序名、主程序名、主程序参数同名

* 函数伪代码

1. **bool** checkIsTheSameAsKey(string id, **int** lineNumber) {
2. 遍历主符号表的第0个到第amount+4个记录{
3. **if** (标识符与记录中的id同名) {
4. **if** (与主程序名同名)
5. 调用addGeneralErrorInformation函数添加通用错误信息;
6. **else** **if** (主程序参数同名)
7. 调用addGeneralErrorInformation函数添加通用错误信息;
8. **else**
9. 调用addGeneralErrorInformation函数添加通用错误信息;
10. **return** **true**;
11. }
12. }
13. **return** **false**;
14. }

#### 找到第X个形式参数的类型

* 接口

string findXthFormalParaType(int X)

* 返回值

string

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| int X | 第X维形式参数 |

* 函数功能

找到第X维形式参数的类型

* 函数伪代码

1. string \_SymbolRecord::findXthFormalParaType(**int** X)
2. {
3. 检查当前flag是否为"function"或者"procedure"，并且传入的维度值未越界；
4. 若是，则返回recordList第x个元素的type值；
5. 若不是，则该记录不是"function"也不是"procedure"，或者是维度值越界；
6. }

#### 检查第X个形式参数是否是引用调用

* 接口

bool isXthFormalParaRefered(int X)

* 返回值

bool

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| int X | 第X维形式参数 |

* 函数功能

检查第X维形式参数是否是引用调用

* 函数伪代码

1. **bool** \_SymbolRecord::isXthFormalParaRefered(**int** X)
2. {
3. 检查当前flag是否为"function"或者"procedure"，并且传入的维度值未越界；
4. 若是，则检查返回recordList第x个元素的flag值是否为"var parameter"，
5. 若是，则返回ture;
6. 若不是，则返回false;
7. 若不是，则该记录不是"function"也不是"procedure"，或者是维度值越界；
8. }

#### 检查第X维下标是否越界

* 接口

bool checkArrayXthIndexRange(int X,int index)

* 返回值

bool

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| int X | 第X维形式参数 |
| int index | 第X个形式参数的下标 |

* 函数功能

检查第X维下标是否越界，true表示越界，false表示未越界

* 函数伪代码

1. **bool** \_SymbolRecord::isXthFormalParaRefered(**int** X)
2. {
3. 检查当前flag是否为"array",并且传入的维度值未越界；
4. 若是，则检查第x个元素的下标值是否大于数组下界并且小于数组上界，
5. 若两者都符合，返回ture;
6. 否则，返回false;
7. 若不是，则该记录不是"array",或者是维度值越界；
8. }

#### 检查id是否是引用参数

* 接口

bool checkIsReferedPara(\_SymbolTable\* currentSymbolTable, string id)

* 返回值

bool

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_SymbolTable\* currentSymbolTable | 指向当前符号表的指针 |
| string id | 标识符id |

* 函数功能

检查检查id是否是引用参数

* 函数伪代码

1. **bool** checkIsReferedPara(\_SymbolTable\* currentSymbolTable, string id) {
2. **if** (当前符号表为主符号表)
3. **return** **false**;
4. **if** (当前符号表的id列表有值) {
5. 根据id列表的下标找到记录在符号表中的下标;
6. 从符号表中找到该记录;
7. **return** 该记录的类型赋值为"var parameter";
8. }
9. **else** {
10. cout << "[checkIsReferedPara] id not found" << endl;
11. }
12. }

#### 符号表指针

\_SymbolTable \*mainSymbolTable;

指向主符号表的指针；

extern \_SymbolTable \*currentSymbolTable;

指向当前符号表的指针。

### 符号表定位与重定位

#### 定位

在进入到抽象语法树每个子程序的定义节点时，创建子符号表，将程序名、参数个数、子符号表指针等信息保存到主符号表后，再定位到子符号表，继续进行程序定义部分的语义分析。

#### 重定位

在退出抽象语法树某个子程序的定义节点时，需要重定位到主符号表。

## 库程序支持

我们将支持read、write、writeln、exit四个库程序调用。

read用于读取内容到指定的变量中，实参只能是普通变量、数组元素或参数，不能是常量或表达式，且不支持读取boolean类型的变量。

write和writeln都用于输出内容到标准输出，后者比前者在最后多输出一个换行符，不同于read，实参可以是各种变量、常量、表达式，且支持boolean类型的输出。

exit相当于C中的return，可以带0个实参，用于过程的返回，也可以带1个实参，用于函数的返回值，后者要求实参表达式与函数返回值类型一致。

库程序在作用域检查中，始终处于最关键的地位。即任何标识符都不能和库程序同名。因此，库程序需要提前加入主符号表中，相当于主符号表中的“钉子户”。正如1.1主程序分析中的主符号表存储结构，这四个库程序固定的处于amount+1到amount+4的位置，amount指主程序的参数个数。

## 作用域检查

### 作用域规则

PASCAL-S采用最内嵌套的静态作用域规则，即：由一个定义引进的标识符在这个定义所在的作用域里可见，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的对同名标识符的另一个定义所掩盖。**由于PASCAL-S不支持程序的嵌套定义，且变量和常量的定义只能在程序一开始的位置，在复合语句块之前，所以其作用域层次最多只有两层。**

### 重定义检查

在一个标识符被定义时，需要进行重定义检查，包括：主程序头中的主程序名、主程序参数；子程序头中的子程序名、子程序参数；主程序中的变量和常量定义；子程序中的变量和常量定义。

重定义检查中，我们规定了三种无视任何作用域不能与其同名的标识符，即库程序名、主程序名、主程序参数名，也就是说，即使是在子程序中定义的参数、变量等标识符也不能和这三种标识符同名。

下面先介绍几种重定义检查的特殊情况，最后再对重定义的检查范围进行分析。

#### 库程序名、主程序名、主程序参数名的重定义检测

地位最高的是库程序名，主程序名和主程序参数名都不能和库程序名同名。

其次是主程序名，所有的主程序参数名除了不能和库程序同名，也不能和主程序名同名。

最后是主程序参数名，除了要求不能和库程序名、主程序名同名，还要求主程序参数名之间不能重名。

其它所有的符号都不能和这三者同名，所以专门设计了一个函数，用于在进行其它重定义检测前，先检测与这三者的同名情况。该函数的介绍在1.7.1.2中。

#### 子程序名

首先需要检查是否和库程序名、主程序名、主程序参数名同名。

过程/函数除了拥有自己的子符号表，其本身的头信息都保存在主符号表中，因此在碰到子程序定义时，需检测子程序名是否和主符号表中的已有标识符重命名。

#### 子程序内部定义符号

首先需要检查是否和库程序名、主程序名、主程序参数名同名。

子程序内部定义的符号不能和子程序名同名，包括参数、变量、常量定义，这也就解释了为什么需要将子程序名保存在子符号表的第0个位置。

#### 检查范围

3.1中已经说明了PASCAL-S所采用的作用域规则，因此，在检测当前定义标识符是否存在重定义的问题时，除了需要检测其是否与库程序名、主程序名、主程序参数名同名，只需要检测其是否与当前符号表内已有的符号重定义即可，无需再返回到外层符号表进行检查。

### 未定义检查

在一个标识符被引用而非定义时，需要对其进行未定义检查，这是进行进一步的类型检查的前提。

未定义检查不像重定义检查有那么多的特殊情况，只需要从当前符号表检查是否已定义，如果当前符号表无定义，且当前符号表是子符号表，还需返回到外层主符号表进行检查。

另外，需要特别注意**循环变量的未定义检查和常量定义右值标识符的未定义检查**。

## 语法成分检查

**该部分检查是为了弥补语法分析，对一些含义不明的语法成分进行更加具体的分类，便于后续语义分析和代码生成**。

### \_VariantReference种类检查

首先需要在\_VariantReference的节点定义中添加string kind域，表示变量引用的具体类型，还添加了int locFlag域，当取值为-1时，表示当前变量引用出现在了赋值语句的左侧，其它情况均用取值为1来表示。

由于语法分析时，无法得知语义信息，所以许多语法成分都被归到了\_VariantReference节点类中。**可能是变量、常量、数组元素引用、函数调用时作为右值、函数返回值中作为左值**。所以kind的取值情况如下：

|  |  |
| --- | --- |
| variantReference的种类 | kind取值 |
| 变量 | var |
| 常量 | constant |
| 数组元素 | array |
| 函数无参调用，作为右值 | function call |
| 函数返回值语句，作为左值 | function return reference |

当\_Expression的type取值为“var”时，该表达式就是由\_VariantReference构成的最简单的表达式，即退化为了\_VariantReference的情形。

### \_Statement是否是返回语句

扩充\_Statement的类定义，加入bool isReturnStatement域。

#### 赋值语句

如果赋值语句的左值是当前所在的函数名，即对左值\_VariantReference进行语义检查，发现其是当前所在的函数名，则该赋值语句是一条返回值语句。

#### 过程调用

如果是exit过程调用，则该过程调用语句就是返回语句。出现在过程定义中时，exit应不带实参；出现在函数中时，exit应带一个实参。

## 数量一致性检查

### 数组维数

在使用数组元素时，需保证数组的下标表达式列表的大小与其定义时的维数相同，其定义时的维数需要通过查符号表获得。

### 子程序调用参数个数

#### 用户自定义程序

在调用程序时，需保证实参表达式列表的大小与该程序定义时的形参个数相同，其定义时的形参的个数需要通过查符号表获得。

#### 库程序

* read至少有一个实参表达式
* write至少有一个实参表达式
* writeln可以有任意个实参表达式，也就是说，可以无参调用，作用相当于输出只输出一个换行符
* exit在过程定义中出现时，必须无参调用；在函数中出现时，必须有且只有一个实参表达是

### 符号种类错误检查

在引用某一符号时，首先需要检查是否已定义，如果已定义，还需**检查已定义的符号的种类是否能够用于当前引用的情景**，例如已定义的real型变量就不能作为循环变量来使用，**这种类型的错误的检查在整个语义分析中占了较大的篇幅**。

以下列举了各种符号种类错误的情况：

* 主程序参数名、过程名作为变量引用，即作为赋值语句的左值，或者作为表达式的操作数
* 函数名作为左值，但是不是当前所在函数名，无法作为返回值语句
* 当前变量引用包含下标表达式列表，也就是作为数组元素来引用，但是之前的定义不是数组
* 只有普通变量、传值参数、引用参数可以作为循环变量（且类型必须是integer），其它种类的符号如果作为循环变量，均应报错。通常还应考虑数组元素作为循环变量的可能性，但是我们的文法不支持这种情况
* 常量作为左值
* 非过程名作为过程调用语句
* 引用形参对应的实参、read的实参只能是普通变量、传值参数、引用参数、数组元素，不能是别的种类
* 常量定义时，可以用别的常量标识符来定义当前常量的取值，但不能是别的种类的标识符
* 出现在表达式中的函数调用，其名称并不是函数名的情况

## 类型检查

### 表达式类型检查

#### \_Expression节点扩展

为\_Expression节点类分别添加一个表示表达式类型的域

1. //语义分析相关
2. **public**:
3. string expressionType;//区别于type，这个值表示表达式的具体类型，即int、real、char、bool、error，其中error表示表达式中包含类型不一致的操作数等情况

#### \_VariantReference节点扩展

为\_VariantReference节点类分别添加一个表示变量、常量或数组元素类型的域

1. **public**:
2. string variantType;//int、real、char、bool、error，其中error表示数组某一维下标表达式的类型不为"int"或标识符不存在

#### \_FunctionCall节点扩展

为\_FunctionCall节点类分别添加一个表示返回值类型的域

1. **public**:
2. string returnType;//int、real、char、bool、error，其中error表示函数标识符不存在

#### \_Expression节点类型分析

|  |  |
| --- | --- |
| 节点类型 | 分析 |
| “var” | 其取值由\_VariantReference类表示，有可能是变量、常量或者数组。获取\_VariantReference的类型即为当前var的类型。注意当\_VariantReference表示数组时，当某一维的下标表达式类型不为”int”时，其类型应设为”error”，但是返回给其父\_Expression节点时，应返回查符号表得到的元素类型。 |
| “int” | 返回”int”。 |
| “real” | 返回”real”。 |
| “function” | 其返回值由\_FunctionCall类表示，获取\_FunctionCall的类型即可。 |
| “compound”:: “relop” | 关系运算符，如果两个子表达式类型一致，且不为error，或者其中一方类型为”int”，另一方类型为”real”，则返回”bool”；否则返回”error”。 |
| “compound”:: “NOT” | 取非运算符，如果子表达式类型为”bool”，则返回”bool”；否则返回”error”。 |
| “compound”:: “minus” | 取相反数运算符，如果子表达式类型为”int”，则返回”int”；如果子表达式类型为”real”，则返回”real”；否则返回”error”。 |
| “compound”:: “bracket” | 直接调用子表达式的获取类型的方法。 |
| “compound”:: ”+” || ”-” || ”\*” || ”/” | 算术运算符，如果两个子表达式类型一致，且不为error，则类型为子表达式的类型；如果其中一方类型为”int”，另一方类型为”real”，则类型为”real”；否则类型为”error”。 |
| “compound”:: ”div” || ”mod” | 如果两个子表达式类型均为”int”，则类型为“int”；否则类型为”error”。 |
| “compound”:: ”and” || ”or” | 如果两个子表达式类型均为”bool”，则类型为”bool”；否则类型为”error”。 |
| \_FunctionCall节点类型分析 | 检查函数调用各个实际参数与符号表中形式参数的一致性（也仅支持int到real的隐式类型转换）  查符号表，如果没有查到，则为error;查到但不为function，也为error;查到且为function，返回其类型 |

#### \_VariantReference节点类型分析

如果是数组，则需检查各维下标表达式的类型是否为"int"

如果是变量或常量，则需查符号表，方法类似functionCall

### 语句的类型检查

在程序设计语言中，像语句、语句序列、程序等这样的语法成分是没有数据类型的，这里所说的语句的类型检查是一种**广义的类型检查**，即，如果这些结构中不含有类型错误，就把回避类型void指派给它；如果在语句中发现了类型错误，则将类型error指派给它。

#### AST语句类扩展

\_Statement基类添加语句类型域，各派生类通过继承获得该类型域。

1. string statementType;//区别于type，取值为"void"或"error"

#### 各种语句类型分析

|  |  |
| --- | --- |
| 语句类型 | 分析 |
| 赋值语句 | 检查左边变量类型和右边表达式类型是否相同（或有隐式转换），不匹配则报错 |
| 过程调用语句 | ①首先检查符号表中是否有该函数/过程，如果没有则报错；  ②如果有检查参数的个数以及各参数的类型与其对应的符号表中形参类型是否匹配，如果不匹配则报错 |
| if条件语句 | ①检查表达式和语句的类型是否为error，如果有任何一个是error则报错  ②检查if后的表达式是否为bool，不是则报错 |
| for循环语句 | ①检查终值、初值是否均为int，如果不是则报错  ②检查语句类型是否为error，如果是则报错 |
| while循环语句 | ①检查表达式的类型是否为bool，如果不是则报错  ②检查语句的类型是否为error，如果是则报错 |
| repeat循环语句 | ①检查表达式是否有类型错误以及语句类型是否为error，如果有则报错  ②检查until后的表达式类型是否为bool，不是则报错 |
| compound语句 | 检查vector中每一条Statement是否有类型错误，任意一条有错则报错，由于compound语句是继承statement的子类，因此要调用子类的属性需要将statement类型的参数利用reinterpret\_cast函数转化成compound类之后，再对其进行语义分析。 |

### 程序调用参数类型检查

#### 用户自定义程序的调用

遇到用户自定义程序的调用时，在形参与实参个数相同的前提下，只需要检查对应位置的形参和实参的类型是否相同，实参的类型通过对实参类型表达式的语义分析获得，形参的类型则通过查符号表的固定位置即可（参见1符号表设计），即第1到第amount个位置存放的是当前程序所定义的所有参数。

#### 库程序的调用

read在实参个数和种类为变量（普通变量、参数、数组元素）的情况下，要求其类型不能为boolean。

write和writeln对实参表达式没有任何限制。

exit在函数定义中出现时，作为该函数的返回值语句，要求其唯一的实参表达式的类型与当前所在函数定义的返回值类型一致。（按照前文，直接找到当前符号表最前面的记录，就可以找到当前函数定义的类型）。

另外，还必须强调传值参数和引用参数的区别。当形式参数为传值参数时，支持实参为integer，形参为real的隐式类型转换，但当形式参数为引用参数时，类型必须强一致，不支持隐式类型转换。

### 函数返回值类型检查

如果检测到当前语句是返回值语句，需检查提供的返回值表达式与函数返回值类型是否一致，函数返回值语句的两种表现形式前文已提及，不再赘述。

### 数组下标表达式类型检查

引用数组元素时，其下标应为integer类型，所以需检查数组下标表达式是否为integer类型。

### 循环变量类型检查

for语句中的循环变量应检查是否为integer类型。

## 涉及常量的语义分析

### 常量定义时的常数传播

常数传播指的是常量右值由别的常量标识符定义，无法直接取得其取值的情况。

#### 常数传播的目标

在引用常量时，能够直接从符号表中查到常量的绝对值（无符号值）和其符号情况（正负）。

#### 涉及的数据结构说明

* 抽象语法树的\_Constant节点

|  |  |
| --- | --- |
| 域定义 | 描述 |
| string type | 常量类型，包括”id”、”integer”、”real”、”char”，其中”id”表示该常量的取值由别的常量标识符提供，右值常量标识符也由下述strOfVal提供，当然在进行符号种类检查之前，并不能确定该标识符是已定义的常量标识符 |
| string strOfVal | 常量取值的字符串表示 |
| bool isMinusShow | 右值前是否出现负号（出现了正号和没有出现任何符号的意义是一样的，即没有出现负号） |

**这里的isMinusShow的含义只能理解为，是否需要在右值的基础上取相反数，并不能代表当前常量取值的正负性**。

* 符号表的常量类型

当符号表中的符号类型为常量时，涉及到以下域：

|  |  |
| --- | --- |
| 域定义 | 描述 |
| type | 常量类型，包括“integer”、”real”、”char” |
| string value | 常量取值 |
| bool isMinusShow | 是否是负数，对于“char”类型，该域没有意义 |

这里的isMinusShow的含义也可以理解为是否需要在value的基础上取相反数，但我们规定常数传播的目标是，value中保存的一定是无符号数，所以**isMinusShow应该深入理解为当前常量是否为负数**。

#### 常量传播方法

如果碰到当前常量的右值由别的常量标识符定义，首先应该检查右值是否未定义；如果已定义，还需检查右值是否是常量标识符；如果是常量标识符，则需从符号表中查出其无符号取值value和符号情况isMinusShow。

|  |  |  |
| --- | --- | --- |
| 当前\_Constant节点的isMinusShow 记为A | 符号表查出的isMinusShow 记为B | 当前常量存入符号表后的isMinusShow 记为C |
| true（取反） | true（负数） | false（非负数） |
| true（取反） | false（非负数） | true（负数） |
| false（不取反） | true（负数） | true（负数） |
| flase（不取反） | false（非负数） | false（非负数） |

观察可得，C=A异或

### 计算整型常量表达式的值

为了便于进行后续的除0错误、数组下标越界等检查，我们支持整型常量表达式的自动计算。

#### 涉及的数据结构说明

常量表达式并不会保存到符号表中，所以要想保存常量表达式的值，只能保存在抽象语法树的\_Expression节点中，涉及到以下域：

|  |  |
| --- | --- |
| 域定义 | 描述 |
| int totalIntValue | 如果该表达式的类型为整型常量，则该域表示其整型取值 |
| bool totalIntValueValid | 表示该表达式的类型是否为整型常量 |
| string expressionType | 该表达式的类型，取值为”integer”、”real”、”char”、”boolean”，也可能是“error”，表示类型检查发现错误 |

#### 标记整型常量表达式和计算取值的方法

|  |  |
| --- | --- |
| \_Expression种类 | 标记和计算方法 |
| \_Expression的种类为”var”，且其附属\_VariantReference经过检查，种类为”constant”，且其类型为“integer” | 从符号表查出常量取值和符号情况，组合成一个有符号整型数  totalIntValueValid=true;  totalIntValue=有符号整型数; |
| \_Expression的种类为”integer” | totalIntValueValid=true;  totalIntValue=当前表达式节点保存的整型数 |
| \_Expression的种类为”compound”，且运算符种类为”minus”；且操作数表达式的类型为”integer”且其totalIntValueValid取值为true | totalIntValueValid=true;  totalIntValue=操作数表达式的常量取值的相反数 |
| \_Expression的种类为”compound”，且运算符种类为”bracket”；且操作数表达式的类型为”integer”且其totalIntValueValid取值为true | totalIntValueValid=true;  totalIntValue=操作数表达式的常量取值 |
| \_Expression的种类为“compound”，且运算符为+、-、\*、/、div、mod；且两个操作数的表达式的类型均为”integer”，totalIntValueValid取值均为true | totalIntValueValid=true;  totalIntValue=左操作数表达式的取值 操作符 右操作数表达式的取值 |

未出现的\_Expression类型，则不涉及整型常量表达式的计算。

### 除0错误的静态检查

涉及/、div、mod三种运算符，如果右操作数表达式是常量整型表达式，就可以判断其取值是否为0，如果为0，就可以静态检测出除0错误。

### 数组下标越界的静态检查

如果数组下标表达式是常量整型表达式，就可以通过查符号表中的数组定义的当前维上下界，检查该维下标是否越界。

### 数组定义时上下界大小关系

根据文法，数组定义时的上下界均为无符号数，而不会是常量表达式的情形，所以直接比较上界是否大于下界即可。

## 函数返回值语句存在性检测（警告信息）

函数如果没有返回值语句，则其返回值将会是一个随机的值，许多编译器都将其作为一个警告检测，我们也简单的实现了一个函数返回值语句的存在性检测，如果检测到有函数没有或缺少返回值语句，则输出警告信息。

### 算法分析

按照文法定义，函数定义中的语句体一定是一个复合语句块\_Compound。\_Compound中的任何一条语句是返回值语句或“完全”地包含返回值语句，就可以认为该函数定义存在足够的返回值语句。\_Compound的语句列表中的每一条语句，具体类型可能各不相同，不同的语句对于返回值语句的考虑也不太一样。

我们首先定义一个概念，“返回值完备”，下面是对其的递归定义：

* 如果一条语句是赋值形式的返回值语句或者exit过程调用形式的返回值语句，就认为该语句是“返回值完备”的
* 如果\_Compound语句中，有任意一条语句是“返回值完备”的，那么该\_Compound语句就是“返回值完备”的
* 如果\_RepeatStatement、\_WhileStatement、\_ForStatement的循环体语句是“返回值完备”的，那么该语句就是“返回值完备”的
* 如果\_IfStatement包含then语句和else语句，且这两条语句均是“返回值完备”的，那么该\_IfStatement语句就是“返回值完备”的
* 如果函数定义的\_Compound程序体是“返回值完备”的，那么该函数定义是“返回值完备”的

下面以表格的形式进行阐述：

|  |  |
| --- | --- |
| 语句种类 | 分析 |
| \_Compound | 语句列表中，只要存在有任何一条语句是返回值语句或“完全”地包含返回值语句，就可以认为该\_Comound语句是“返回值完备”的。 |
| \_RepeatStatement | 如果循环体语句是“返回值完备”的，就可以认为该\_RepeatStatement语句是“返回值完备”的。 |
| \_WhileStatement | 如果循环体语句是“返回值完备”的，就可以认为该\_RepeatStatement语句是“返回值完备”的。 |
| \_ForStatement | 如果循环体语句是“返回值完备”的，就可以认为该\_RepeatStatement语句是“返回值完备”的。 |
| \_IfStatement | 如果\_IfStatement不包含else部分，那么无论then语句是否是“返回值完备”，该\_IfStatement语句都不是“返回值完备”的；如果\_IfStatement包含else部分，那么当then语句和else语句都是“返回值完备”时，就可以认为该\_IfStatement语句是“返回值完备”的，否则就认为该­\_IfStatement语句不是“返回值完备”的。 |
| \_AssignStatement | 如果是赋值语句形式的函数返回值语句，就认为该\_AssignStatement语句是“返回值完备”的 |
| \_ProcedureCall | 如果是exit过程调用形式的函数返回值语句，就认为该\_ProcedureCall语句是“返回值完备”的 |

### 数据结构

涉及的数据结构只有保存警告信息的列表，声明如下：

extern vector<string> semanticWarningInformation;

其定义在SemanticAnalyse.cpp中，因此这里需要用extern关键字声明。

### 警告信息格式设计

* 需要提供函数名functionId和函数名所在的行号functionLineNumber
* 格式为：[Return value statement missing!] <Line {functionLineNumber}> Incomplete return value statement of function "{functionId}".

### 引用外部的函数接口

#### 整型数字转化为字符串

* 函数接口

extern string itos(int num);//将整数转化为字符串

### 内部函数接口

#### 检查当前语句是否是“返回值完备”的

* 函数接口

bool returnExistedCheckStatement(\_Statement \*statementNode);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Statement \*statementNode | 语句基类指针 |

* 返回值

bool，true表示当前语句是“返回值完备”的，false表示当前语句不是“返回值完备的”

* 伪代码

bool returnExistedCheckStatement(\_Statement \*statementNode) {

    if (statementNode == NULL) {

        cout << "[returnExistedCheckStatement] pointer of \_Statement is null" << endl;

        return false;

    }

    if (是compound语句) {

        \_Compound \*compound = reinterpret\_cast<\_Compound\*>(statementNode);

        遍历语句列表 {

调用returnExistedCheckStatement检查当前语句的“返回值完备”性;

if(当前语句是返回值完备的)

return true;

        }

        return false;

    }

    else if (是repeat语句) {

        \_RepeatStatement \*repeatStatement = reinterpret\_cast<\_RepeatStatement\*>(statementNode);

        调用returnExistedCheckStatement检查循环体语句的“返回值完备”性;

if(循环体语句是“返回值完备”的)

return true;

else

return false;

    }

    else if (是while语句) {

        \_WhileStatement \*whileStatement = reinterpret\_cast<\_WhileStatement\*>(statementNode);

        调用returnExistedCheckStatement检查循环体语句的“返回值完备”性;

if(循环体语句是“返回值完备”的)

return true;

else

return false;

    }

    else if (是for语句) {

        \_ForStatement \*forStatement = reinterpret\_cast<\_ForStatement\*>(statementNode);

        调用returnExistedCheckStatement检查循环体语句的“返回值完备”性;

if(循环体语句是“返回值完备”的)

return true;

else

return false;

    }

    else if (是if语句) {

        \_IfStatement \*ifStatement = reinterpret\_cast<\_IfStatement\*>(statementNode);

        if (ifStatement->then == NULL) {

            cout << "[returnExistedCheckStatement][IfStatement] the pointer of then statement is null";

            return false;

        }

        if (if语句不包含else部分)

            return false;

        else{

            调用returnExistedCheckStatement检测then语句和else语句的“返回值完备”性;

if(then语句和else语句都是“返回值完备”的)

return true;

else

return false;

}

    }

    else if (是赋值语句) { //判断是否是返回值语句

        \_AssignStatement \*assignStatement = reinterpret\_cast<\_AssignStatement\*>(statementNode);

        return assignStatement->isReturnStatement; //在之前的语义分析中，已经保存是否是返回值语句

        //即fun:=expression的情况

    }

    else if (是过程调用) { //判断是否是返回值语句

        \_ProcedureCall \*procedureCall = reinterpret\_cast<\_ProcedureCall\*>(statementNode);

        return procedureCall->isReturnStatement; //在之前的语义分析中，已经保存是否是返回值语句

        //即exit()的情况

    }

    else {

        cout << "[returnExistedCheckStatement] statement type error" << endl;

        return false;

    }

}

#### 检查函数定义的程序体语句列表是否是“返回值完备”的

* 函数接口

void returnExistedCheckMainCompound(\_Compound \*compoundNode, string functionId, int functionLineNumber);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Compound \*compoundNode | 程序体语句列表 |
| string functionId | 函数名 |
| int functionLineNumber | 函数名所在的行号 |

* 返回值

无

* 伪代码

void returnExistedCheckMainCompound(\_Compound \*compoundNode, string functionId, int functionLineNumber) {

    if (compoundNode == NULL) {

        cout << "[returnExistedCheckCompound] pointer of \_Compound is null" << endl;

        return;

    }

    bool ok = false; //表示当前复合语句块的“返回值完备”性，true表示具有“返回值完备”性，false表示不具有“返回值完备”性

    遍历复合语句块的语句列表 {

调用returnExistedCheckStatement检测当前语句的“返回值完备”性;

if(当前语句是返回值完备的){

ok=true;

break;

}

    }

    if (复合语句块不是“返回值完备”的) {

按照函数缺少返回值语句的警告信息格式组织functionId、functionLineNumber，形成一条警告信息;

将警告信息添加到警告信息列表中;

    }

}

#### 检查函数定义是否是“返回值完备”的

* 函数接口

void returnExistedCheckFunctionDefinition(\_FunctionDefinition\* functionDefinitionNode);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_FunctionDefinition\* functionDefinitionNode | 函数定义节点类指针 |

* 返回值

无

* 伪代码

void returnExistedCheckFunctionDefinition(\_FunctionDefinition\* functionDefinitionNode) {

    if (functionDefinitionNode == NULL) {

        cout << "[returnExistedCheckFunction] pointer of \_FunctionDefinition is null" << endl;

        return;

    }

    调用returnExistedCheckMainCompound检查函数定义的程序体是否是“返回值完备”的;

}

## 错误检查和恢复策略详述

### 恢复策略

#### 形参重定义

形参发生重定义错误时，不能只是报错，因为后续还会可能会调用子程序，在调用时，需要进行形参和实参的一致性检查，如果在函数定义时，没有成功将形参加入符号表，会对后续的调用检查产生影响。

所以这里需要进行适当的恢复，恢复的策略就是仍将形参加入符号表，但需要进行一些改动。考虑到PASCAL-S的标识符不包含下划线，所以我们可以在形参和别的符号发生重名时，在其前面加入下划线。而且形参之间可能还会再次重名，所以需要统计重名的次数，然后根据重名的次数来决定在标识符前面加多少个下划线，最后再加入符号表即可。

#### 主程序名重定义

当主程序名和库程序同名时，采用和形参重定义相同的加下划线的策略。

#### 其它标识符的重定义

其它标识符在检测到重定义时，无需进行恢复，因为我们无从得知程序员接下来对该标识符的使用是按照哪种定义的。

#### 类型检查

* 数组引用时，即使其下标表达式类型检查出错，或者下标表达式取值越界，其检查类型的返回值仍为其定义时的数组元素类型
* 函数调用时，即使其调用的语义分析检测出错误，其检查类型的返回值仍为其定义时的返回值类型

#### 数组定义时下界大于上界

修改上界为下界值，再存入符号表即可

### 各抽象语法树的语义分析策略详述

#### \_VariantReference

注意，检查成功的情况，**需要标记\_VariantReference的具体种类**，即是”array”、”var”、“constant”、”function call”还是”function return reference”

* 检测是否未定义，**若未定义，则放弃当前节点的后续检查**
* 若已定义，且是非数组引用
  + 如果是当前所在函数名
    - 检查是否是函数返回值赋值语句的左值
    - 否则就是函数的无参递归调用，需检查函数定义时形参个数是否为0
  + 如果是函数名
    - 检查是否是赋值语句的左值，是就报错
    - 否则就是函数的无参调用，需检查函数定义时形参个数是否为0
  + 如果是函数名、传值参数、引用参数、普通变量、常量以外的情况，报错
* 若已定义，且是数组引用
  + 如果定义不是数组，则报错
  + 如果定是数组，则检查下标维数是否一致，表达式类型是否都是整型，如果表达式常量整型表达式，还应进行越界的静态检查

#### \_Satatement

* 如果是复合语句块
  + 对语句列表中的每一个语句进行语义分析
* 如果是repeat语句
  + 检查条件表达式的类型
  + 对循环语句体进行语义分析
* 如果是while语句
  + 检查条件表达式的类型
  + 对循环语句体进行语义分析
* 如果是for语句
  + 检查循环变量
    - **如果未定义，放弃for语句的后续语义分析**
    - **如果种类、类型错误，放弃for语句的后续语义分析**
  + 检查初值表达式和终值表达式的类型
  + 对循环语句体进行语义分析
* 如果是if语句
  + 检查条件表达式的类型
  + 对then语句体进行语义分析
  + 如果有else部分，对else语句体进行语义分析
* 如果是赋值语句
  + 对左值\_VariantReference和右值\_Expression进行语义分析
    - **如果左值是常量，则放弃赋值语句的后续语义分析**
    - 如果是左值函数名，且作为当前函数的返回值语句，则检查右值表达式的类型是否和当前函数返回值类型一致
    - 否则检查左值和右值的类型是否一致
* 如果是过程调用语句
  + **如果过程名未定义，则放弃当前过程调用语句的后续语义分析**
  + **如果定义的种类不是过程，则放弃当前过程调用语句的后续语义分析**
  + 如果是exit过程调用
    - 如果当前所在程序是过程，则检查exit的实参是否为0
    - 如果当前所在程序是函数，则检查exit的实参是否为1，类型是否和函数返回值一致，对实参表达式进行语义分析
  + 如果是read过程调用，需检查实参个数是否至少为1，检查实参的种类是否正确，且类型不能为boolean；对实参表达式进行语义分析
  + 如果是write过程调用，需检查实参个数是否至少为1；对实参表达式进行语义分析
  + 如果是用户自定义过程调用，需检查实参和形参的一致性，如果形参是引用参数，实参就必须是普通变量、传值参数、引用参数、数组元素，而不能是别的情况；对实参表达式进行语义分析

#### \_FormalParameter

* 如果当前参数名和库程序名、主程序名或主程序参数名同名，报错并**恢复**
* 如果当前参数名和当前符号表中已定义的符号同名，报错并**恢复**
* 将形式参数添加到符号表中

#### \_FunctionDefinition

* **如果重定义，则放弃当前函数/过程定义的后续语义分析**
* 将函数/过程定义加入到主符号表中，创建并定位到子符号表
* 对形式参数列表进行语义分析
* 对常量定义列表进行语义分析
* 对变量定义列表进行语义分析
* 对语句列表进行语义分析
* 如果是函数，对语句列表进行函数返回值语句存在性检测

#### \_Variant

* **如果重定义，则放弃当前变量定义的后续语义分析**
* 如果是普通变量，则加入符号表
* 如果是数组，则检查是否存在下界大于上界的情况，如果存在，进行报错和**恢复**，最后加入符号表

#### \_Constant

* **如果重定义，则放弃当前常量定义的后续语义分析**
* 如果直接由常量提供取值，则加入符号表
* 如果由别的标识符提供取值，则检查右值标识符是否未定义，如果已定义，检查是否是常量，如果都没有问题，就加入符号表

#### \_SubProgram

* 对全局常量定义列表进行语义分析
* 对全局变量定义列表进行语义分析
* 对子程序定义列表进行语义分析，每个子程序分析完后，都要重定位到主符号表
* 对主程序语句列表进行语义分析，由于主程序可以看做是一个过程，所以无需进行函数返回值语句存在性检测

#### \_FunctionCall

* **如果函数名未定义，则报错，放弃当前函数调用的后续语义分析**
* **如果定义的不是函数名，则报错，放弃当前函数调用的后续语义分析**
* 检查实参与形参的一致性，对实参表达式进行语义分析，如果形参是引用参数，还需检查实参是否是普通变量、传值参数、引用参数或数组元素。

#### \_Expression

* 如果是”var”
  + 对\_VariantReference进行语义分析
  + 检查是否是整型常量，如果是，计算取值，并标记为整型常量表达式
  + 返回类型\_VariantReference的类型作为当前表达式的类型
* 如果是”integer”
  + 标记为整型常量表达式
  + 返回”integer”类型
* 如果是”real”
  + 返回“real”类型
* 如果是“char”
  + 返回”char”类型
* 如果是“function”
  + 对\_FunctionCall进行语义分析
  + 返回\_FunctionCall的类型作为当前表达式的类型
* 如果是“compound”：：“relop”
  + 对两个操作数表达式进行语义分析
  + 对两个操作数表达式的类型进行检查
  + 返回”boolean”或“error”类型
* 如果是“compound”：：“not”
  + 对操作数表达式进行语义分析
  + 对操作数表达式的类型进行检查
  + 返回“boolean”或“error”类型
* 如果是“compound”：：“minus”
  + 对操作数表达式进行语义分析
  + 检查、计算、标记整型常量表达式
  + 对操作数表达式的类型进行检查
  + 返回”integer”或“real”或“error”类型
* 如果是“compound”：：“bracket”
  + 对操作数表达式进行语义分析
  + 返回操作数表达式的类型作为当前表达式的类型
* 如果是“compound”：：“+、-、\*、/”
  + 对两个操作数表达式进行语义分析
  + 检查、计算、标记整型常量表达式
  + 对两个操作数表达式的类型进行检查
  + 返回“integer”或“real”或“error”类型
* 如果是“compound”：：“div、mod”
  + 对两个操作数表达式进行语义分析
  + 检查、计算、标记整型常量表达式
  + 对两个操作数表达式的类型进行检查
  + 返回”integer”或“error”类型
* 如果是“compound”：：“and”、or”
  + 对两个操作数表达式进行语义分析
  + 对两个操作数表达式的类型进行检查
  + 返回”boolean”或”error”类型

#### \_Program

* 检查主程序名是否与库程序名同名，如果同名，报错并进行**恢复**，并将信息加入主符号表
* 检查主程序参数名是否与库程序名、主程序名同名，如果同名，报错并进行**恢复**，并将信息加入主符号表
* 将库程序加入主符号表
* 对\_SubPorgam进行语义分析

## 警告信息处理

### 存储警告信息的数据结构

警告信息均存储于以下的警告信息列表中：

vector<string> semanticWarningInformation

### 警告信息格式

本编译器只支持一种警告信息，即函数缺少返回值语句的情形，所以也未设置任何处理警告信息格式的函数。在检测出函数缺少返回值语句时，直接按照下面的格式添加警告信息：

1. warningInformation = "[Return value statement missing!] <Line " + itos(functionLineNumber) + "> Incomplete return value statement of function \"" + functionId + "\".";
2. semanticWarningInformation.push\_back(warningInformation);

其中functionLineNumber指的是函数定义的函数名所在的行号，functionId指的是函数名。

## 错误信息处理

### 存储错误信息的数据结构

错误信息均存储于以下的错误信息列表中：

vector<string> semanticErrorInformation

### 添加错误信息函数

添加错误信息的函数分为两种。

一种是具体的，具有明确用途的添加错误函数，只要给这些函数提供必要的信息，它就可以帮你将这些信息组装成一条具体针对某一错误的报错信息，该种函数主要针对在语义分析的过程中重复出现的错误，也可以是将某些错误归为一类，再提供诸如用途之类的信息，就可以统一使用一个接口来添加错误信息。

另一种，或者说一个，是直接把提供的字符串加入错误信息列表的函数，主要用于那些仅出现一次的错误类型，无需封装成单独的函数。

#### 重定义

* 接口

void addDuplicateDefinitionErrorInformation(string preId, int preLineNumber, string preFlag, string preType,int curLineNumber)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string preId, | 之前已定义的标识符id |
| int preLineNumber | 之前已定义的标识符所在行号 |
| string preFlag | 之前已定义的标识符的种类 |
| string preType | 之前已定义的标识符的类型 |
| int curLineNumber | 当前标识符的行号 |

* 函数功能

获得重复定义的语义错误信息。

* 输出格式：

与其他标识符同名时，输出：

1. errorInformation += "\"" + preId + "\"" + " has already been defined as a " + preFlag + " at line " + itos(preLineNumber) + ".";

与主程序名、主程序参数名、库程序名同名时，输出：

1. errorInformation += "\"" + preId + "\"" + " has already been defined as a lib program.";

#### 未定义

* 接口

void addUndefinedErrorInformation(string id, int curLineNumber)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 标识符的id |
| int curLineNumber | 标识符所在行号 |

* 函数功能

获得未定义的语义错误信息。

* 输出格式：

1. errorInformation = "[Undefined identifier!] <Line " + itos(curLineNumber) + "> ";
2. errorInformation += id + " has not been defined.";

#### 标识符类型错误

* 接口

void addUsageTypeErrorInformation(string curId, int curLineNumber, string curType, string usage, string correctType)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string curId | 当前标识符的id |
| int curLineNumber | 当前标识符所在行号 |
| string curType | 当前标识符类型 |
| string usage | 之前标识符的位置 |
| string correctType | 之前标识符的类型 |

* 函数功能

标识符类型只能为为基本类型："integer","real","char","boolean"

该函数能获得标识符类型错误的语义错误信息。

* 输出格式：

1. errorInformation = "[Usage type error!] <Line " + itos(curLineNumber) + "> ";
2. errorInformation += "\"" + curId + "\"" + " used for " + usage + " should be " + correctType + " but not " + curType + ".";

#### 数组下标、参数个数不匹配

* 接口

void addNumberErrorInformation(string curId, int curLineNumber, int curNumber, int correctNumber, string usage)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string curId | 当前标识符的id |
| int curLineNumber | 当前标识符所在行号 |
| int curNumber | 当前参数个数 |
| int correctNumber | 正确的参数个数 |
| string usage | 之前标识符的位置 |

* 函数功能

获得数组下标不匹配、函数或过程参数个数不匹配的语义错误信息。

* 输出格式：

数组下标个数不匹配时：

1. errorInformation += "[Array index number mismatch!] ";
2. errorInformation += "<Line " + itos(curLineNumber) + "> ";
3. errorInformation += "Array \"" + curId + "\"" + " should have " + itos(correctNumber) + " but not " + itos(curNumber) + " indices.";

过程参数个数不匹配时：

1. errorInformation += "[Procedure parameter number mismatch!] ";
2. errorInformation += "<Line " + itos(curLineNumber) + "> ";
3. errorInformation += "Procedure \"" + curId + "\"" + " should have " + itos(correctNumber) + " but not " + itos(curNumber) + " parameters.";

函数参数个数不匹配时：

1. errorInformation += "[Function parameter number mismatch!] ";
2. errorInformation += "<Line " + itos(curLineNumber) + "> ";
3. errorInformation += "Function \"" + curId + "\"" + " should have " + itos(correctNumber) + " but not " + itos(curNumber) + " parameters.";

#### 标识符种类错误

* 接口

void addPreFlagErrorInformation(string curId, int curLineNumber, string curFlag, int preLineNumber, string preFlag)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string curId | 当前标识符的id |
| int curLineNumber | 当前标识符所在行号 |
| string curFlag | 当前标识符种类 |
| int preLineNumber | 之前标识符行数 |
| string preFlag | 之前标识符的种类 |

* 函数功能

标识符种类有以下几种："value parameter"表示传值参数,"var parameter"表示传引用参数,"normal variant"表示普通变量,"constant"表示常量,，"array"表示数组,"procedure"表示过程,"function"表示函数，"(sub)program name"表示该条记录是当前符号表对应的程序名信息，"parameter of program"表示主程序的参数，该程序将添加标识符种类错误信息。

* 输出格式：

当标识符种类错误时，输出如下信息：

1. errorInformation += "[Symbol kinds mismatch!] ";
2. errorInformation += "<Line " + itos(curLineNumber) + "> ";
3. errorInformation += "\"" + curId + "\"" + " defined at line " + itos(preLineNumber) + " is a " + preFlag + " but not a " + curFlag + ".";

#### 表达式类型错误

* 接口

void addExpressionTypeErrorInformation(\_Expression \*exp, string curType, string correctType, string description)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*exp | 指向该表达式的指针 |
| string curType | 当前表达式的类型 |
| string correctType | 表达式正确的类型 |
| string description | 表达式作用的描述 |

* 函数功能

表达式类型主要有以下几种,"var"表示变量,"integer"表示整数,"real"表示浮点数,"char"表示常量字符，"function"表示函数调用,"compound"表示复合表达式,compound有普通的二目运算符，还有"minus"、"not"、"bracket"等单目运算符。该函数将添加表达式类型错误信息。

* 输出格式：

1. errorInformation += "Expression \"" + expression + "\" used for " + description + " should be " + correctType + " but not " + curType + ".";

#### 赋值语句两边类型不匹配错误

* 接口

void addAssignTypeMismatchErrorInformation(\_VariantReference \*leftVariantReference, \_Expression \*rightExpression)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_VariantReference \*leftVariantReference | 左值的参数引用列表的指针 |
| \_Expression \*rightExpression | 指向表达式右值的指针 |

* 函数功能

添加赋值语句左值和右值类型不匹配错误信息。

* 输出格式：

左值和右值类型不匹配时，输出：

1. errorInformation += "[Assign statement type mismatch!] ";
2. errorInformation += "<Left at line " + itos(leftVariantReference->variantId.second) + ", right at line " + itos(rightExpression->lineNumber) + "> ";
3. errorInformation += "Left \"" + varRef + "\" type is " + leftVariantReference->variantType + " while right \"" + exp + "\" type is " + rightExpression->expressionType + ".";

#### 数组下标越界

* 接口

void addArrayRangeOutOfBoundErrorInformation(\_Expression \*expression, string arrayId, int X, pair<int,int> range)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*exp | 指向该表达式的指针 |
| string arrayId | 数组id |
| int X | 数组第X维 |
| pair<int,int> range | 数组各维上下界 |

* 函数功能

添加数组下标越界错误信息。

* 输出格式：

数组上下界越界,输出如下信息：

1. errorInformation += "[Array range out of bound!] ";
2. errorInformation += "<Line " + itos(expression->lineNumber) + "> ";
3. errorInformation += "The value of expression \"" + exp + "\"" + " is " + itos(expression->totalIntValue);
4. errorInformation += ", but the range of array \"" + arrayId + "\" " + itos(X) + "th index is " + itos(range.first) + " to " + itos(range.second) + ".";

#### 数组下界比上界大

* 接口

void addArrayRangeOutOfBoundErrorInformation(\_Expression \*expression, string arrayId, int X, pair<int,int> range)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*exp | 指向该表达式的指针 |
| string arrayId | 数组id |
| int X | 数组第X维 |
| pair<int,int> range | 数组各维上下界 |

* 函数功能

添加数组下界比上界大的错误信息。

* 输出格式：

数组下界比上界大时，输出如下信息：

1. errorInformation += "[Array range upsidedown error!] ";
2. errorInformation += "<Line " + itos(curLineNumber) + "> ";
3. errorInformation += itos(X) + "th range of array \"" + curId + "\" have larger low bound and smaller high bound, which is " + itos(lowBound) + " and " + itos(highBound) + ".";

#### 运算符两边的操作数类型不一致

* 接口

void addOperandExpressionsTypeMismatchErrorInformation(\_Expression \*exp1, \_Expression \*exp2)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*exp1 | 指向表达式左值的指针 |
| \_Expression \*exp2 | 指向表达式右值的指针 |

* 函数功能

添加运算符两边的操作数类型不一致的错误信息。

* 输出格式：

当运算符两边的操作数类型不一致时，输出如下信息：

1. errorInformation += "[Operands expression type mismatch!] ";
2. errorInformation += "<Left at line " + itos(exp1->lineNumber) + ", right at line " + itos(exp2->lineNumber) + "> ";
3. errorInformation += "Left \"" + expStr1 + "\" type is " + exp1->expressionType + " while right " + "\"" + expStr2 + "\" type is " + exp2->expressionType + ".";

#### 某个操作数类型错误

* 接口

void addSingleOperandExpressionTypeMismatchErrorInformation(\_Expression \*exp, string correctType)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*exp | 指向表达式的指针 |
| string correctType | 操作数的正确类型 |

* 函数功能

添加某个操作数类型错误的信息。如mod操作符左边标识符类型为real。

* 输出格式：

当某个操作数类型错误时，输出如下信息：

1. errorInformation += "[Operand expression type error!] ";
2. errorInformation += "<Line " + itos(exp->lineNumber) + "> ";
3. errorInformation += "Expression \"" + expStr + "\" type should be " + correctType + " but not " + exp->expressionType + ".";

#### read函数参数出现错误

* 接口

void addactualParameterOfReadErrorInformation(int curLineNumber, string procedureId, int X, \_Expression \*exp)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| int curLineNumber | 过程所在行号 |
| string procedureId | 过程的id |
| int X | read函数的第X个参数 |
| \_Expression \*exp | 指向表达式的指针 |

* 函数功能

添加read的实参错误信息。

* 输出格式：

当read函数参数出现错误,输出如下信息：

1. errorInformation += "[Actual parameter of read procedure type error!] ";
2. errorInformation += "<Line " + itos(curLineNumber) + "> ";
3. errorInformation += "\"" + procedureId + "\" " + itos(X) + "th expression parameter \"" + expression + "\" is not a variant or an array element.";

#### 除0错误

* 接口

void addDivideZeroErrorInformation(string operation, \_Expression \*exp)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string operation | 操作符id |
| \_Expression \*exp | 指向该表达式的指针 |

* 函数功能

添加除0错误信息。

* 输出格式：

当出现除0错误时，输出如下信息：

1. errorInformation += "[Divide zero error!] ";
2. errorInformation += "<Line " + itos(exp->lineNumber) + "> ";
3. errorInformation += "The value of expression \"" + expression + "\" is 0, which is the second operand of operation \"" + operation + "\".";

#### read读取boolean类型错误

* 接口

void addReadBooleanErrorInformation(\_Expression \*exp, int X)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*exp | 指向该表达式的指针 |
| int X | read函数的第X维参数 |

* 函数功能

添加read读取boolean类型变量错误的信息。

* 输出格式：

当出现read读取boolean类型错误时,输出如下信息：

1. errorInformation = "[Read boolean error!] ";
2. errorInformation += "<Line " + itos(exp->lineNumber) + "> ";
3. errorInformation += "The " + itos(X) + "th actual parameter of read \"" + expression + "\" is boolean, it can't be read.";

#### 直接添加错误

* 接口

void addGeneralErrorInformation(string errorInformation)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string id | 程序名的id |
| int lineNumber | 程序名所在行号 |
| string subprogramType | 该program的类型，只有Function和Procedure两种 |
| int amount | 该program的参数个数 |
| string returnType | 该program的返回值类型，若为Procedure，其返回值类型为void |

* 函数功能

添加除上述提及错误之外其他错误的信息。

* 调用该函数的错误场景与输出格式：
  + 程序名与主程序名相同时，输出以下信息：

1. addGeneralErrorInformation("[Invalid reference] <Line " + itos(variantReference->variantId.second) + "> Procedure name \"" + record->id + "\" can't be referenced");
   * 变量引用类型为函数，作为左值时，输出以下信息：
2. addGeneralErrorInformation("[Invalid reference!] <Line " + itos(variantReference->variantId.second) + "> function name \"" + record->id + "\" can't be referenced as l-value.");
   * 变量引用类型不是传值参数、不是传引用参数、不是常量、不是普通变量时，输出以下信息：
3. addGeneralErrorInformation("[Invalid reference!] <Line " + itos(variantReference->variantId.second) + "> \"" + variantReference->variantId.first + "\" is a " + record->flag + ", it can't be referenced.");
   * 当左值为常量时，输出以下信息：
4. addGeneralErrorInformation("[Invalid reference!] <Line " + itos(variantReference->variantId.second) + "> \"" + variantReference->variantId.first + "\" is a " + record->flag + ", it can't be referenced.");
   * 当语句返回值和函数参数类型不一致时，输出以下信息：
5. addGeneralErrorInformation("[Return type of funciton mismatch!] <Line " + itos(assignStatement->expression->lineNumber) + "> The type of return expression is " + rightType + " ,but not " + assignStatement->variantReference->variantType + " as function \"" + assignStatement->variantReference->variantId.first + "\" defined.");
   * 如果exit过程所在程序为过程，exit过程参数不为0，输出以下信息：
6. addGeneralErrorInformation("[Return value redundancy!] <Line " + itos(procedureCall->procedureId.second) + "> Number of return value of procedure must be 0, that is, exit must have no actual parameters.");
   * 如果exit过程所在程序为函数，exit过程参数为0，输出以下信息：
7. addGeneralErrorInformation("[Return value missing!] <Line " + itos(procedureCall->procedureId.second) + "> Number of return value of function must be 1, that is, exit must have 1 actual parameters.");
   * 如果exit过程所在程序为函数，exit过程参数不为1，输出以下信息：
8. addGeneralErrorInformation("[Return value redundancy!] <Line " + itos(procedureCall->procedureId.second) + "> Number of return value of function must be 1, that is, exit must have 1 actual parameters.");
   * 如果exit过程所在程序为函数，exit过程参数为1，但与与函数返回值类型不一致，输出以下信息：
9. addGeneralErrorInformation("[Return type of funciton mismatch!] <Line " + itos(procedureCall->actualParaList[0]->lineNumber) + "> The type of return expression is " + returnType + " ,but not " + currentSymbolTable->recordList[0]->type + " as function \"" + currentSymbolTable->recordList[0]->id + "\" defined.");
   * 如果read、write的参数个数为0，输出以下信息：
10. addGeneralErrorInformation("[" + tmp + " actual parameter missing!] <Line " + itos(procedureCall->procedureId.second) + "> procedure \"" + record->id + "\" must have at least one actual parameter.");
    * 当表达式不能作为引用形参对应的实参时，输出以下信息：
11. addGeneralErrorInformation("[Referenced actual parameter error!] <Line " + itos(procedureCall->actualParaList[i]->lineNumber) + "> The " + itos(i + 1) + "th actual parameter expression should be a normal variable、value parameter、referenced parameter or array element.");
    * 当标识符与主程序同名时，输出以下信息：
12. addGeneralErrorInformation("[Duplicate defined error!] <Line " + itos(program->programId.second) + "> Name of program \"" + program->programId.first + "\" has been defined as a lib program.");
    * 当标识符与主程序参数同名时，输出以下信息：
13. addGeneralErrorInformation("[Duplicate defined error!] <Line " + itos(program->programId.second) + "> parameter of program \"" + program->programId.first + "\" is the same as name of program.");
    * 当标识符与库程序同名时，输出以下信息：
14. addGeneralErrorInformation("[Dulicate defined error!] <Line " + itos(program->paraList[i].second) + "> parameter of program \"" + program->paraList[i].first + "\" has been defined as a lib program.");

## 语义分析函数设计

### 常量语义分析

* 函数接口

void SemanticAnalyseConst(\_Constant\* constant)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Constant\* constant | 指向该常量标识符的指针 |

* 函数功能

对常量定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseConst(\_Constant\* constant){
2. **if**(constant==NULL){
3. cout << "[SemanticAnalyseConst] pointer of \_Constant is null" << endl;
4. **return**;
5. }
6. 用constId.first去查符号表，检查是否重定义
7. 调用checkIsTheSameAsKey函数，检查是否与主程序名、主程序参数名、库程序名同名；
8. 若是，**return**;
9. **if** (符号表记录不为空) {
10. 调用addDuplicateDefinitionErrorInformation函数添加重定义错误信息;
11. **return**;
12. }
13. **if**(该常量由另外的常量标识符定义){
14. 调用findSymbolRecord函数查询该常量标识符的记录;
15. **if**(该常量标识符未定义){
16. 调用addUndefinedErrorInformation函数添加未定义错误信息;
17. **return**;
18. }
19. **if** (该标识符不是常量) {
20. 调用addPreFlagErrorInformation函数添加标识符种类错误信息;
21. **return**;
22. }
23. 调用addConst函数将该标识符添加到符号表记录中;
24. }
25. **else**//该常量由常数值定义
26. 调用addConst函数将该标识符添加到符号表记录中;
27. }

### 变量语义分析

* 函数接口

void SemanticAnalyseVariant(\_Variant\* variant)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Constant\* constant | 指向该变量标识符的指针 |

* 函数功能

对变量定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseVariant(\_Variant\* variant){
2. **if**(variant==NULL){
3. cout << "[SemanticAnalyseVariant] pointer of \_Variant is null" << endl;
4. **return**;
5. }
6. 用variantId.first去查符号表，检查是否重定义；
7. 调用checkIsTheSameAsKey函数，检查是否与主程序名、主程序参数名、库程序名同名；
8. 若是，**return;**
9. **if**(符号表记录不为空){
10. 调用addDuplicateDefinitionErrorInformation添加重定义错误信息；
11. **return**;
12. }
13. **if**(如果当前变量不是数组)
14. 调用addVar将该变量添加进符号表；
15. **else**{
16. **if**(数组每一维上界均大于等于下界)
17. 调用addArray函数将该数组添加进符号表；
18. **else**
19. 调用addArrayRangeUpSideDownErrorInformation添加数组越界错误信息；
20. }
21. }

### 语句语义分析

* 函数接口

void SemanticAnalyseStatement(\_Statement \*statement)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Statement \*statement | 指向该语句标识符的指针 |

* 函数功能

对语句定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseStatement(\_Statement \*statement){
2. **if**(statement==NULL){
3. cout << "[SemanticAnalyseStatement] pointer of \_Statement is null" << endl;
4. **return**;
5. }
6. **if**(语句类型是"compound"){
7. 调用**reinterpret\_cast**函数将该语句转换为\_CompoundStatement类型；
8. 遍历对复合语句块中的每一条语句，调用SemanticAnalyseStatement函数进行语义分析；
9. }
10. **else** **if**(语句类型是"repeat"){
11. 调用**reinterpret\_cast**函数将该语句转换为\_RepeatStatement类型；
12. 调用SemanticAnalyseExpression得到条件表达式的类型；
13. **if** (条件表达式的类型 != "boolean") {//repeat语句类型检查,condition表达式类型检查 checked
14. 调用addExpressionTypeErrorInformation函数添加类型错误信息；
15. 该语句类型赋值为"error"；
16. }
17. **else**
18. 该语句类型赋值为"void";
19. 调用SemanticAnalyseStatement函数对循环体语句进行语义分析；
20. }
21. **else** **if**(语句类型是"while"){
22. 调用**reinterpret\_cast**函数将该语句转换为\_WhileStatement类型；
23. 调用SemanticAnalyseExpression得到条件表达式的类型；
24. string type = SemanticAnalyseExpression(whileStatement->condition);
25. **if** (条件表达式的类型 != "boolean") {//repeat语句类型检查,condition表达式类型检查 checked
26. 调用addExpressionTypeErrorInformation函数添加类型错误信息；
27. 该语句类型赋值为"error"；
28. }
29. **else**
30. 该语句类型赋值为"void";
31. 调用SemanticAnalyseStatement函数对循环体语句进行语义分析；
32. }
33. **else** **if**(语句类型是"for"){
34. 调用**reinterpret\_cast**函数将该语句转换为\_ForStatement类型；
35. 调用 findSymbolRecord函数，找到循环变量的记录；
36. **if**(记录不为空){
37. 调用addUndefinedErrorInformation函数添加重定义错误信息；
38. **return**;
39. }
40. **if**(该记录类型不为传值参数、传引用参数、普通变量){}
41. 调用addPreFlagErrorInformation添加标识符种类错误信息;
42. **return**;
43. }
44. **if** (循环变量类型不为整型) {
45. 调用addUsageTypeErrorInformation函数添加标识符类型错误信息;
46. **return**;
47. }
48. //for语句类型检查,start和end表达式类型检查
49. forStatement语句类型赋值为"void";
50. string type = SemanticAnalyseExpression(forStatement->start);
51. 调用SemanticAnalyseExpression函数获得forStatement语句的start值的类型；
52. **if** (start值的类型不为 "integer") {
53. 调用addUsageTypeErrorInformation函数添加标识符类型错误信息;
54. forStatement语句类型赋值为 "error";
55. }
56. 调用SemanticAnalyseExpression函数获得forStatement语句的end值的类型；
57. **if** (end值的类型不为 "integer") {
58. 调用addUsageTypeErrorInformation函数添加标识符类型错误信息;
59. forStatement语句类型赋值为 "error";
60. }
61. 调用SemanticAnalyseStatement函数对循环体语句进行语义分析
62. }
63. **else** **if**(语句类型是"if"){
64. 调用**reinterpret\_cast**函数将该语句转换为\_IfStatement类型;
65. 调用SemanticAnalyseExpression函数获得条件表达式的类型;
66. **if** (条件表达式的类型不为 "boolean") {
67. 调用addUsageTypeErrorInformation函数添加标识符类型错误信息;
68. ifStatement语句类型赋值为 "error";;
69. }
70. **else**
71. ifStatement语句类型赋值为"void";
72. 调用 SemanticAnalyseStatement对then语句进行语义分析；
73. **if**(**if**语句对应的els不等于NULL)
74. 调用SemanticAnalyseStatement函数对**else**语句进行语义分析;
75. }
76. **else** **if**(语句类型是"assign"){
77. 调用**reinterpret\_cast**函数将该语句转换为\_AssignStatement类型;
78. assign语句类型赋值为"void";
79. 将assign语句的左值的locFlag赋值为-1；
80. 调用SemanticAnalyseVariantReference函数获得assign语句的左值的类型；
81. **if** (assign语句的左值的类型为 "constant") {
82. 调用addGeneralErrorInformation函数添加通用错误信息;
83. **return**;
84. }
85. 调用SemanticAnalyseExpression函数获得assign语句的右值的类型
86. **if** (assign语句的右值的类型为 "function return reference") {
87. **if** (assign语句的右值的类型不等于函数返回值类型 && !(assign语句的右值的类型为"integer"且函数返回值类型为"real")) {
88. 调用addGeneralErrorInformation函数添加通用错误信息;
89. assign语句类型赋值为"error";
90. }
91. assign语句是返回值语句；
92. **return**;
93. }
94. //比较左值和右值类型,获得赋值语句的类型；类型不同时，只支持整型到实型的隐式转换
95. **if** (左值类型不等于右值类型 && !(左值类型=="real" && 右值类型=="integer")) {
96. 调用addAssignTypeMismatchErrorInformation函数添加左值类型与右值类型不匹配的错误信息;
97. assign语句赋值为 "error";
98. }
99. **else**
100. assign语句类型赋值为"void";
101. }
102. **else** **if**(语句类型是"procedure"){//read的参数只能是变量或数组元素;
103. 调用**reinterpret\_cast**函数将该语句转换为\_ProcedureCall类型;
104. 通过procedureId，调用findSymbolRecord函数查主表，获得这条语句的记录;
105. **if** (记录为空)
106. 通过procedureId，调用findSymbolRecord函数查当前符号表，获得这条语句的记录;
107. 将procedure语句类型赋值为 "void";
108. **if** (记录为空) {
109. 调用addUndefinedErrorInformation函数添加重定义错误信息；;
110. 将procedure语句类型赋值为"error";
111. **return**;
112. }
113. **if** (该记录类型不是 "procedure") {
114. 调用addPreFlagErrorInformation函数添加标识符种类错误信息;
115. 将procedure语句类型赋值为"error";
116. **return**;
117. }
118. **if** (语句类型是 "exit") {
119. **if** (当前程序是"procedure") {
120. **if** (exit语句的实参个数 != 0) {
121. 调用addGeneralErrorInformation函数添加通用错误信息;
122. exit语句类型赋值为"error";
123. }
124. **return**;
125. }
126. //如果是函数
127. **if** (exit语句的实参个数 != 1) {
128. **if** (exit语句的实参个数== 0)
129. 调用addGeneralErrorInformation函数添加通用错误信息;
130. **else**
131. 调用addGeneralErrorInformation函数添加通用错误信息;
132. **return**;
133. }
134. 调用SemanticAnalyseExpression函数获得函数返回值的类型
135. **if** (实参表达式的类型 !=函数返回值的类型 && !(实参表达式的类型为 "real" && 函数返回值的类型为"integer")) {
136. 调用addGeneralErrorInformation函数添加通用错误信息;
137. 将procedure语句类型赋值为"error";
138. }
139. procedure语句是返回值语句；
140. **return**;
141. }
142. **if** (记录的id类型为 "read" || 记录的id类型为"write") {
143. **if** (procedure语句的参数为0) {
144. 调用addGeneralErrorInformation函数添加通用错误信息;
145. 将procedure语句类型赋值为"error";
146. }
147. }
148. **if** (记录的id类型为 "read") {
149. 遍历read函数中的每一个参数，调用SemanticAnalyseExpression得到read函数各个参数的参数类型；
150. **if** (!(read函数的参数类型引用域type为 "var" && (read函数的参数类型为"var" || read函数的参数类型为 "array")))
151. 调用addactualParameterOfReadErrorInformation函数来添加read的实参错误信息;
152. **if** (read函数的参数表达式类型 == "boolean")
153. 调用addReadBooleanErrorInformation函数来添加添加read读取boolean类型变量错误的信息;
154. **if** (read函数的参数类型== "error")
155. read语句的类型赋值为 "error";
156. }
157. **return**;
158. }
159. **if** (记录的amount值 == -1) {
160. 遍历对语句块中的每一条语句，调用SemanticAnalyseExpression函数得到read函数各个参数的参数类型{
161. **if** (read函数的参数类型 == "error")
162. read语句类型赋值为"error";
163. }
164. **return**;
165. }
166. **if** (语句的参数个数!= 记录的语句参数个数) {
167. 调用addNumberErrorInformation函数添加参数个数不匹配错误新;
168. read语句类型赋值为"error";
169. **return**;
170. }
172. 遍历对语句块中的每一条语句，调用SemanticAnalyseExpression函数得到read函数的各个参数的参数类型 {
173. 调用findXthFormalParaType函数，得到对应的每一个实参的参数类型；
174. 调用isXthFormalParaRefered函数来得到每一个实参的类型是否是传引用调用；
175. **if** (实参类型是传引用调用 && !(语句变量调用变量值为"var" && (实参传引用调用表达式类型为 "var" || 实参传引用调用表达式类型为"array"))) {
176. 调用addGeneralErrorInformation函数添加通用错误信息;
177. **continue**;
178. }
179. **if** (实参类型是传值调用) {
180. **if** (实参类型与形参类型不一致 && !(实参类型为"integer" && 形参类型为"real")) {
181. 调用addExpressionTypeErrorInformation函数添加类型错误信息;
182. procedureCall->statementType = "error";
183. }
184. }
185. **else** {
186. **if** (实参类型与形参类型不一致) {
187. 调用addExpressionTypeErrorInformation函数添加类型错误信息;
188. procedureCall->statementType = "error";
189. }
190. }
191. }
192. }
193. **else** {
194. cout << "[SemanticAnalyseStatement] statement type error" << endl;
195. **return**;
196. }
197. }

### 表达式语义分析

* 函数接口

string SemanticAnalyseExpression(\_Expression\* expression)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression\* expression | 指向该表达式标识符的指针 |

* 函数功能

对表达式定义进行语义分析

* 函数伪代码

1. string SemanticAnalyseExpression(\_Expression\* expression) {
2. **if**(expression==NULL){
3. cout << "[SemanticAnalyseExpression] pointer of \_Expression is null" << endl;
4. **return** "";
5. }
6. **if** (表达式类型为 "var") {
7. 调用SemanticAnalyseVariantReference函数获得常量的类型;
8. **if** (常量引用类型为"integer" && 该常量类型为 "constant") {
9. 调用findSymbolRecord函数来查符号表查出常量值
10. **if** (记录不存在) {
11. cout << "[SemanticAnalyseExpression] pointer of record is null" << endl;
12. **return** "";
13. }
14. **if** (记录的类型!= "constant") {
15. cout << "[SemanticAnalyseExpression] the record should be a constant" << endl;
16. **return** "";
17. }
18. 将记录的常数值赋值给该表达式的值;
19. **if** (表达式的值为负值)
20. 将记录的值加上负号之后赋值给表达式的值;
21. 常量表达式的值存在;
22. }
23. **return** 表达式的类型;
24. }
25. **else** **if** (表达式类型为 "integer") {
26. 将表达式的整数值赋值给该表达式的值;
27. 整型表达式的值存在;
28. **return** 表达式的具体类型为 "integer";
29. }
30. **else** **if** (表达式类型为 "real")
31. **return** 表达式的类型为 "real";
32. **else** **if** (表达式类型为  "char")
33. **return**表达式的类型为 "char";
34. **else** **if** (表达式类型为"function")
35. **return** 调用SemanticAnalyseFunctionCall函数获得函数调用的返回值类型;
36. **else** **if** (表达式类型为"compound") {
37. **if** (表达式的操作类型为 "relop") {
38. 调用SemanticAnalyseExpression函数获得表达式左值的类型;
39. 调用SemanticAnalyseExpression函数获得表达式右值的类型;
40. **if** ((表达式左值的类型不等于表达式右值的类型 && 表达式左值的类型不为 "error") || (表达式左值的类型为 "integer" && 表达式右值的类型为 "real") || (表达式左值的类型为"real" &&表达式右值的类型为"integer"))
41. **return**  表达式的具体类型为 "boolean";
42. **else**{
43. **if**(表达式左值的类型不等于表达式右值的类型 &&表达式左值的类型不等于"error" &&表达式左值的类型不等于 "error")
44. 调用addOperandExpressionsTypeMismatchErrorInformation函数来添加操作符两边类型不一致错误信息；
45. **return** 表达式的具体类型为 "error";
46. }
47. }
48. **else** **if** (表达式的操作类型为 "not") {
49. 调用SemanticAnalyseExpression函数获得表达式的类型；
50. **if** (表达式的类型为"boolean")
51. **return** 表达式的具体类型 "boolean";
52. **else**{
53. **if**(表达式的类型不为 "error" && 表达式的类型不为"boolean")
54. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
55. **return** 表达式的具体类型为 "error";
56. }
57. }
58. **else** **if** (表达式的操作类型为 "minus") {
59. 调用SemanticAnalyseExpression函数获得表达式的类型;
60. **if**(表达式的类型为"integer" && 表达式操作数的值存在){
61. 将表达式操作数的值取负之后，重新赋值为表达式的值;
62. 表达式的值存在;
63. }
64. **if** (表达式的类型为"integer" || 表达式的类型为 "real")
65. **return** 返回表达式的具体类型;
66. **else**{
67. **if**(表达式的类型不为 "error" &&表达式的类型不为"integer" &&表达式的类型不为"real")
68. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
69. **return** 表达式的具体类型为 "error";
70. }
71. }
72. **else** **if** (表达式的操作类型为"bracket"){
73. 调用SemanticAnalyseExpression函数获得表达式的类型;
74. **if**(表达式的类型为 "integer" && 表达式操作数的值存在){
75. 将表达式操作数的值赋值为表达式的值;
76. 表达式的值存在;
77. }
78. **return** 表达式的类型;
79. }
80. **else** **if** (表达式的操作类型为 "+" ||表达式的操作类型为 "-" || 表达式的操作类型为 "\*" || 表达式的操作类型为"/") {
81. 调用SemanticAnalyseExpression函数获得表达式第一个操作数的类型;
82. 调用SemanticAnalyseExpression函数获得表达式第二个操作数的类型;
83. **if** (表达式的操作类型为"/" && 除数类型为"integer" &&除数的值存在&& 除数的值为 0)
84. 调用addDivideZeroErrorInformation函数添加除0错误信息;
85. **if**(第一个操作数类型为"integer" &&第二个操作数的类型为 "integer" && 第一个操作数的值存在 && 第二个操作数的值存在){
86. 表达式的值存在;
87. **if**(表达式的操作类型为"+")
88. 表达式的值等于第一个操作数的值+第二个操作数的值;
89. **else** **if**(表达式的操作类型为"-")
90. 表达式的值等于第一个操作数的值-第二个操作数的值;
91. **else** **if**(表达式的操作类型为"\*")
92. 表达式的值等于第一个操作数的值\*第二个操作数的值;
93. **else**
94. 表达式的值等于第一个操作数的值/第二个操作数的值;
95. }
96. **if**((第一个操作数类型为"integer" || 第一个操作数类型为"real") && (第二个操作数类型为"integer" || 第二个操作数类型为"real")){
97. **if**(第一个操作数类型为"integer" && 第二个操作数类型为"integer")
98. **return** 表达式的类型为"integer";
99. **return** 表达式的类型为"real";
100. }
101. **if**(第一个操作数类型不为 "error" && 第一个操作数类型不为 "integer" && 第一个操作数类型不为"real")
102. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
103. **if**(第二个操作数类型不为"error" && 第二个操作数类型不为 "integer" && 第二个操作数类型不为 "real")
104. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
105. **return** 表达式的类型为"error";
106. }
107. **else** **if** (表达式的操作类型 "div" || 表达式的操作类型"mod") {
108. 调用SemanticAnalyseExpression函数获得表达式第一个操作数的类型;
109. 调用SemanticAnalyseExpression函数获得表达式第二个操作数的类型;
110. **if** (第二个操作数的类型"integer" && 第二个操作数的值存在&& 第二个操作数的值为0)
111. 调用addDivideZeroErrorInformation函数添加除0错误信息;
112. **if** (第一个操作数类型为"integer" && 第二个操作数类型为"integer"){
113. **if**((第一个操作数的值存在 && 第二个操作数的值存在){
114. **if**(表达式的操作类型 "div")
115. 表达式的值等于第一个操作数的值/第二个操作数的值;
116. **else**
117. 表达式的值等于第一个操作数的值%第二个操作数的值;
118. 表达式的值存在;
119. }
120. **return** 表达式的类型为"integer";
121. }
122. **if**(第一个操作数类型不为"error" &&第一个操作数类型不为"integer")
123. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
124. **if**(第二个操作数类型不为"error" && 第二个操作数类型不为"integer")
125. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
126. **return** 表达式的类型为"error";
127. }
128. **else** **if** (表达式的操作类型为 "and" || 表达式的操作类型为"or") {
129. 调用SemanticAnalyseExpression函数获得表达式第一个操作数的类型;
130. 调用SemanticAnalyseExpression函数获得表达式第二个操作数的类型;
131. **if** (第一个操作数类型为"boolean" && 第二个操作数类型为 "boolean")
132. **return** expression->expressionType = "boolean";
133. **if** (第一个操作数类型不为 "error" && 第一个操作数类型不为"boolean")
134. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
135. **if** (第二个操作数类型不为 "error" && 第二个操作数类型不为"boolean")
136. 调用addSingleOperandExpressionTypeMismatchErrorInformation函数添加某个操作数类型错误信息;
137. **return** 表达式的类型为 "error";
138. }
139. **else** {
140. cout << "[\_Expression::SemanticAnalyseExpression] ERROR: operation not found" << endl;
141. **return** "error";
142. }
143. }
144. }

### 程序语义分析

* 函数接口

void SemanticAnalyseProgram(\_Program \*program)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Statement \*statement | 指向该程序标识符的指针 |

* 函数功能

对程序定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseProgram(\_Program \*program) {
2. **if** (program == NULL) {
3. cout << "[SemanticAnalyseProgram] pointer of \_Program is null" << endl;
4. **return**;
5. }
6. 调用lib.insert函数将read、write、writeln、exit 4个库函数加入函数库;
7. **if** (程序与库函数名同名)
8. 调用addGeneralErrorInformation添加通用错误信息;
9. 调用addProgramName函数将该程序的函数名存储到主符号表的第0个位置,将主程序的参数添加到主符号表中，flag定为"parameter of program;
10. 遍历主程序参数{
11. **if** (该程序名是否和主程序名同名)
12. 调用addGeneralErrorInformation函数添加通用错误信息;
13. **else** **if** (该程序名是否和主程序参数名同名)
14. 调用addGeneralErrorInformation函数添加通用错误信息;
15. **else**
16. 调用addVoidPara函数将主程序参数加入到符号表中;
17. }
19. 调用addProcedure函数添加read过程，该过程变参
20. 调用addProcedure函数添加write过程，该过程变参
21. 调用addProcedure函数添加writeln过程，该过程变参
22. 调用addProcedure函数添加exit过程;
23. 调用SemanticAnalyseSubprogram对分程序进行语义分析;
24. }

### 分程序语义分析

* 函数接口

void SemanticAnalyseSubprogram(\_SubProgram\* subprogram)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_SubProgram\* subprogram | 指向该分程序标识符的指针 |

* 函数功能

对分程序定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseSubprogram(\_SubProgram\* subprogram){
2. **if**(subprogram==NULL){
3. cout << "[SemanticAnalyseSubprogram] pointer of \_Subprogram is null" << endl;
4. **return**;
5. }
6. 遍历分程序的常量列表
7. 调用SemanticAnalyseConst函数对常量进行语义分析;
8. 遍历分程序的变量列表
9. 调用SemanticAnalyseVariant函数对变量进行语义分析;
10. 遍历分程序的子程序定义列表{
11. 调用SemanticAnalyseSubprogramDefinition函数对子程序定义进行语义分析;
12. 进行符号表的重定位;
13. }
14. 调用 SemanticAnalyseStatement函数对语句进行语义分析;
15. }

### 子程序语义分析

* 函数接口

void SemanticAnalyseSubprogramDefinition(\_FunctionDefinition\* functionDefinition)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_FunctionDefinition\* functionDefinition | 指向该子程序标识符的指针 |

* 函数功能

对子程序定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseSubprogramDefinition(\_FunctionDefinition\* functionDefinition){
2. **if**(functionDefinition==NULL){
3. cout << "[SemanticAnalyseSubprogramDefinition] pointer of \_FunctionDefinition is null" << endl;
4. **return**;
5. }
6. 调用findSymbolRecord函数查找该子程序定义的记录;
7. **if**(记录不为空){
8. 调用addDuplicateDefinitionErrorInformation函数添加未定义错误信息;
9. **return**;
10. }
11. **if** (子程序无返回值)
12. 子程序的类型为 "procedure";
13. **else**
14. 子程序的类型为"function";
15. 调用 createSubSymbolTableAndInit()函数创建并定位到子符号表;
16. 调用addProgramName函数将子程序名等信息添加到子符号表中;
17. **if** (如果该程序是过程)//如果是过程
18. 调用addProcedure函数将过程添加到主程序表中;
19. **else**
20. 调用addFunction函数将函数添加到主程序表中;
21. 遍历子程序的形参列表{
22. 调用SemanticAnalyseFormalParameter函数对形式参数列表进行语义分析，并将形式参数添加到子符号表中;
23. }
24. 遍历子程序的常量列表{
25. 调用SemanticAnalyseConst函数对常数列表进行语义分析，并将常数添加到子符号表中;
26. }
27. 遍历子程序的变量列表{
28. 调用SemanticAnalyseVariant函数对变量列表进行语义分析，并将变量添加到子符号表中;
29. }
30. 调用SemanticAnalyseStatement函数对compound进行语义分析;
31. 调用returnExistedCheckFunctionDefinition函数对函数进行返回值语句的存在性检查;
32. }

### 函数调用语义分析

* 函数接口

string SemanticAnalyseFunctionCall(\_FunctionCall \*functionCall)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_FunctionCall \*functionCall | 指向该函数调用标识符的指针 |

* 函数功能

对函数调用定义进行语义分析

* 函数伪代码

1. string SemanticAnalyseFunctionCall(\_FunctionCall \*functionCall) {
2. **if**(functionCall==NULL){
3. cout << "[SemanticAnalyseFunctionCall] pointer of \_FunctionCall is null" << endl;
4. **return** "";
5. }
6. 调用findSymbolRecord函数在主符号表查找该函数调用定义的记录
7. **if** (记录为空)
8. 调用findSymbolRecord函数在当前符号表查找该函数调用定义的记录
9. **if** (记录为空) {
10. 调用addaddUndefinedErrorInformation函数添加未定义错误信息;
11. **return** 函数调用类型为 "error";
12. }
13. **if** (记录类型不是"function") {
14. 调用addPreFlagErrorInformation函数添加标识符种类错误信息;
15. **return** 函数调用类型为 "error";
16. }
17. **if** (函数的参数量为-1) {
18. 遍历函数的参数列表
19. 调用SemanticAnalyseExpression函数对每一个参数进行表达式的语义分析;
20. **return** 记录的类型;
21. }
22. **if** (参数个数不一致) {
23. 调用addNumberErrorInformation函数添加参数个数不一致错误信息;
24. **return** 记录的类型;
25. }
26. //检查各位置的实参和形参类型是否一致 形参在符号表中的定位
27. 遍历函数的参数列表 {
28. string actualType = SemanticAnalyseExpression(functionCall->actualParaList[i]);
29. string formalType = record->findXthFormalParaType(i + 1);
30. 调用SemanticAnalyseExpression函数得到每一个实参的类型;
31. 调用findXthFormalParaType函数得到每一个形参的类型;
32. 调用isXthFormalParaRefered函数检查第X维形式参数是否是引用调用;
33. **if** (是引用调用 && !(实参类型引用为 "var" && (实参类型为 "var" || 实参类型为"array"))) {
34. 调用addGeneralErrorInformation函数添加通用错误信息;
35. **continue**;
36. }
37. **if** (是传值参数) {
38. **if** (实参类型与形参不一致&& !(实参类型为 "integer" && 形参类型为 "real"))
39. 调用addExpressionTypeErrorInformation函数添加表达式类型错误信息;
40. }
41. **else** {
42. **if** (实参类型与形参不一致)
43. 调用addExpressionTypeErrorInformation函数添加表达式类型错误信息;
44. }
45. }
46. **return** 函数调用类型;
47. }

### 形式参数语义分析

* 函数接口

void SemanticAnalyseFormalParameter(\_FormalParameter \*formalParameter)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_FunctionCall \*functionCall | 指向该形式参数标识符的指针 |

* 函数功能

对形式参数定义进行语义分析

* 函数伪代码

1. **void** SemanticAnalyseFormalParameter(\_FormalParameter\* formalParameter){
2. **if**(formalParameter==NULL){
3. cout << "[SemanticAnalyseFormalParameter] pointer of \_FormalParameter is null" << endl;
4. **return**;
5. }
6. **if** (调用checkIsTheSameAsKey函数检查是否与库程序名、主程序名、主程序参数同名)
7. **return**;
8. 调用findSymbolRecord函数在当前符号表中寻找该标识符记录;
9. **if**（记录不为空）
10. 调用addDuplicateDefinitionErrorInformation函数添加重定义错误信息;
11. **if**(形参类型传值参数)
12. 调用addPara函数添加传值参数标识符到符号表中;
13. **else**
14. 调用addVarPara函数添加传引用参数标识符到符号表中;
15. }

### 变量引用语义分析

* 函数接口

string SemanticAnalyseVariantReference(\_VariantReference\* variantReference)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_VariantReference\* variantReference | 指向该变量引用标识符的指针 |

* 函数功能

对变量引用定义进行语义分析

* 函数伪代码

1. string SemanticAnalyseVariantReference(\_VariantReference\* variantReference){
2. **if**(variantReference==NULL){
3. cout << "[SemanticAnalyseVariantReference] pointer of \_VariantReference is null" << endl;
4. **return** "";
5. }
6. 调用findSymbolRecord函数在当前符号表查找该变量引用定义的记录
7. **if** (记录为空) {
8. 调用addaddUndefinedErrorInformation函数添加未定义错误信息;
9. **return** 该变量引用类型为 "error";
10. }
11. **if** (该变量引用不是数组) {
12. **if** (记录类型引用为"(sub)program name") {
13. **if** (记录类型为 "procedure") {
14. 调用addGeneralErrorInformation函数添加通用错误信息;
15. **return** 该变量引用类型为"error";
16. }
17. **if** (变量引用为左值) {
18. 变量引用的种类为 "function return reference";
19. **return** 变量引用的类型;
20. }
21. //如果是右值
22. **if** (变量引用为右值且形参个数不为0) {
23. 调用addNumberErrorInformation函数添加参数个数不一致错误信息;
24. **return** 变量引用的类型;
25. }
26. 变量引用的种类为"function call";
27. **return** 变量引用的类型;
28. }
29. **if** (记录类型为 "function") {
30. 变量引用类型为 "function";
31. **if** (变量引用为左值) {
32. **return** 变量引用的类型为 "error";
33. 调用addGeneralErrorInformation函数添加通用错误信息;
34. }
35. **if** (如果形参个数不为0) {
36. addNumberErrorInformation(variantReference->variantId.first, variantReference->variantId.second, 0, record->amount, "function");
37. 调用addNumberErrorInformation函数添加参数个数不一致错误信息;
38. **return** 变量引用的类型;
39. }
40. **return** 变量引用的类型;
41. }
42. **if** (!(记录类型为"value parameter" || 记录类型为 "var parameter" || 记录类型为 "normal variant" || 记录类型为 "constant")) {
43. 调用addGeneralErrorInformation函数添加通用错误信息;
44. **return** 变量引用的类型"error";
45. }
46. 变量引用的种类为 "var";
47. **if** (记录的类型为 "constant")
48. 变量引用的类型赋值为"constant";
49. **return** 变量引用的类型;
50. }
51. **else** **if** (变量引用的类型为数组) {
52. **if** (记录不是 "array") {
53. 调用addPreFlagErrorInformation函数添加标识符种类错误信息;
54. **return** 变量引用的类型为"error";
55. }
56. 变量引用的种类赋值为 "array";
57. **if** (变量引用的下标维数和符号表所存记录不一致) {
58. 调用addNumberErrorInformation函数添加参数个数不一致错误信息;
59. 变量引用的类型赋值为 "error";
60. **return** 变量引用的类型;
61. }
62. 将符号表记录类型赋值给变量引用类型;
63. 遍历变量引用的的表达式列表 {
64. 调用SemanticAnalyseExpression函数得到每一个表达式的类型;
65. **if** (表达式下标类型不为"integer") {
66. variantReference->variantType = "error";
67. 调用addExpressionTypeErrorInformation函数添加表达式类型错误信息;
68. 变量引用的类型赋值为 "error";
69. }
70. **if**(变量引用的值存在){
71. **if**(调用checkArrayXthIndexRange函数检查每一维是否越界){
72. 调用addArrayRangeOutOfBoundErrorInformation函数添加数组下标越界错误信息;
73. 变量引用的类型赋值为 "error";
74. }
75. }
76. }
77. **return** 记录的类型;
78. }
79. **else** {
80. cout << "[SemanticAnalyseVariantReference] flag of variantReference is not 0 or 1" << endl;
81. **return** 变量引用的类型为 "error";
82. }
83. }