# 语义分析详细设计

## 语义分析概述

语义分析是编译的第三个阶段，主要完成的任务是建立符号表和类型检查。在源代码中通常有两种语句：声明语句和可执行语句。

在声明语句中，会声明一些数据对象或过程，并为之取相应的名字，即标识符；因此对于声明语句，语义分析的主要任务就是收集标识符的属性信息和语义检查。标识符的属性包括种属（如变量、常量、过程等）、类型（integer、char等）以及名称等，通过收集这些信息，语义分析程序会建立一张符号表，记录源程序中定义的所有记录；而对声明语句的语义检查，则主要聚焦于是否重定义等；

对可执行语句的语义分析大体可以理解为对可执行语句的语法检查及自动调整，比如：变量名未声明就使用，函数参数不匹配等情况，并在适当的时候进行隐式类型转换，以及输出警告信息；

通过语义分析程序，我们可以收集到源程序中出现的所有语义错误，并建立一张完整的符号表。

## 接口分析

### 与语法分析的接口

与语法分析的接口是语法分析树的根节点，即ParseTreeHead;

**Token**\* ParseTreeHead;

### AST生成接口

* 字符串转整型
  + 函数接口：int **str2int**(**string** str);
  + 参数列表
    - string str ：需要转化的字符串
  + 返回值：转化后的int值
* 字符串转浮点型
  + 函数接口：float **str2float**(**string** str);
  + 参数列表
    - string str ：需要转化的字符串
  + 返回值：转化后的float值
* 获取整个程序
  + 函数接口：**\_Program**\* **getProgram**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：返回AST根结点指针\_Program类指针
* 获取主程序头
  + 函数接口：void **getProgramHead**(**Token** \*now, **pair**<**string**, int>& \_programId, **vector**< **pair**<**string**, int> >& \_paraList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - pair<string, int>& \_programId ：主程序名及行号
    - vector< pair<string, int> >& \_paraList ：参数名及行号列表
  + 返回值：空
* 获取主程序体
  + 函数接口：**\_SubProgram**\* **getProgramBody**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：子程序体，\_SubProgram类指针
* 获取标识符列表
  + 函数接口：void **getIdList**(**Token** \*now, **vector**< **pair**<**string**, int> >& res,bool reverseFlag);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector< pair<string, int> >& res  ：标识符名及行号列表
    - bool reverseFlag ：是否反转标识符列表
  + 返回值：空
* 获取常数值
  + 函数接口：void **setConst**(**Token** \*now, **\_Constant**\* &\_constant);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - \_Constant\* &\_constant ：待修改的常量定义
  + 返回值：空
* 获取常量定义列表
  + 函数接口：void **getConstList**(**Token** \*now, **vector**<**\_Constant**\*>& \_constantList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Constant\*>& \_constantList ：常量定义列表
  + 返回值：空
* 获取常量定义
  + 函数接口：void **getConst**(**Token** \*now, **vector**<**\_Constant**\*>& \_constantList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Constant\*>& \_constantList ：常量定义列表
  + 返回值：空
* 获取重构定义列表
  + 函数接口：void **getTypeDefList**(**Token** \*now,**vector**<**\_TypeDef**\*>& \_typedefList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_TypeDef\*>& \_typedefList ：重构定义列表
  + 返回值：空
* 获取重构定义
  + 函数接口：void **getTypeDef**(**Token** \*now,**vector**<**\_TypeDef**\*>& \_typedefList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_TypeDef\*>& \_typedefList ：重构定义列表
  + 返回值：空
* 获取变量定义列表
  + 函数接口：void **getVariantList**(**Token** \*now,**vector**<**\_Variant**\*>& \_variantList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_TypeDef\*>& \_typedefList ：变量定义列表
  + 返回值：空
* 获取变量定义
  + 函数接口：void **getVariant**(**Token** \*now,**vector**<**\_Variant**\*>& \_variantList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_TypeDef\*>& \_typedefList ：变量定义列表
  + 返回值：空
* 获取类型
  + 函数接口：**\_Type**\* **getType**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：类型具体定义，即\_type类型指针
* 获取数组上下界列表
  + 函数接口：

void **getArrayRangeList**(**Token** \*now,**vector**< **pair**<int,int> >& \_arrayRangeList);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector< pair<int,int>>& \_arrayRangeList：数组各维上下界列表
  + 返回值：空
* 获取数组上下界
  + 函数接口：

void **getArrayRange**(**Token** \*now,**vector**< **pair**<int,int> >& \_arrayRangeList);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector< pair<int,int>>& \_arrayRangeList：数组各维上下界列表
  + 返回值：空
* 获取函数/过程定义列表
  + 函数接口：void **getSubprogramDefinitionList**(**Token** \*now,

**vector**<**\_FunctionDefinition**\*>& \_subprogramDefinitionList);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_FunctionDefinition\*>& \_subprogramDefinitionList ：函数/过程定义列表
  + 返回值：空
* 获取函数/过程定义
  + 函数接口：**\_FunctionDefinition**\* **getSubprogramDefinition**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：子程序定义类指针
* 获取子程序头
  + 函数接口：

void **getSubprogramHead**(**Token** \*now,**pair**<**string**,int>& functionID,

**vector**<**\_FormalParameter**\*>& \_formalParaList,**pair**<**string**,int> &\_type);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - pair<string,int>& functionID ：子程序名及行号
    - vector<\_FormalParameter\*>& \_formalParaList ：参数列表
    - pair<string,int> &\_type ：返回值类型及行号，可能为空串，表示无返回值，即当前子程序是过程
  + 返回值：空
* 获取形参列表
  + 函数接口：void **getFormalParaList**(**Token** \*now,**vector**<**\_FormalParameter**\*>

& \_formalParaList);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_FormalParameter\*>& \_formalParaList ：形式参数列表
  + 返回值：空
* 获取形参
  + 函数接口：void **getFormalParameters**(**Token** \*now,

**vector**<**\_FormalParameter**\*>& \_formalParaList);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_FormalParameter\*>& \_formalParaList ：形式参数列表
  + 返回值：空
* 区别形参
  + 函数接口：void **getParameter**(**Token** \*now,**vector**<**\_FormalParameter**\*>& \_formalParaList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_FormalParameter\*>& \_formalParaList ：形式参数列表
  + 返回值：空
* 获取形参具体信息
  + 函数接口：void **getValueParameter**(**Token** \*now,

**vector**<**\_FormalParameter**\*>& \_formalParaList,int flag);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_FormalParameter\*>& \_formalParaList ：形式参数列表
    - int flag ：flag=0表示传值参数，flag!=0表示传引用参数
  + 返回值：空
* 获取子程序体
  + 函数接口：

void **getSubprogramBody**(**Token** \*now,**vector**<**\_Constant**\*>& \_constList,

**vector**<**\_TypeDef**\*> typedefList,**vector**<**\_Variant**\*>& \_variantList,

**vector**<**\_FunctionDefinition**\*> & subprogramDefinitionList,

**\_Compound**\* &\_compound);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Constant\*>& \_constList ：子程序常量定义
    - vector<\_TypeDef\*> \_typedefList ：子程序重构定义
    - vector<\_Variant\*>& \_variantList ：子程序变量定义
    - vector<\_FunctionDefinition\*> &\_subprogramDefinitionList ：子程序子程序定义
    - \_Compound\* &\_compound ：子程序函数体
  + 返回值：空
* 获取函数体中复合语句
  + 函数接口：**\_Compound**\* **getCompoundStatement**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：复合语句指针
* 获取语句列表
  + 函数接口：

void **getStatementList**(**Token** \*now,**vector**<**\_Statement**\*>& \_statementList);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Statement\*>& \_statementList ：语句列表
  + 返回值：空
* 获取单条语句
  + 函数接口：**\_Statement**\* **getStatement**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：语句类指针
* 获取结构体的属性或数组的维度上下限列表
  + 函数接口：

void **getidVpartList**(**Token** \*now,**vector**<**\_Idvpart**\*> &idvpartlist);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Idvpart\*> &idvpartlist ：引用变量后缀列表
  + 返回值：空
* 获取结构体的属性或数组的维度上下限
  + 函数接口：void **getVpart**(**Token** \*now,**vector**<**\_Idvpart**\*> &idvpartlist);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Idvpart\*> &idvpartlist ：引用变量后缀列表
  + 返回值：空
* 获取过程调用语句
  + 函数接口：**\_Statement**\* **getProcedureCall**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：语句类指针
* 获取表达式列表
  + 函数接口：void **getExpressionList**(**Token** \*now,**vector**<**\_Expression**\*>& \_expressionList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Expression\*>& \_expressionList ：表达式列表
  + 返回值：空
* 获取表达式
  + 函数接口：**\_Expression**\* **getExpression**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：表达式类指针
* 获取简单表达式
  + 函数接口：**\_Expression**\* **getSimpleExpression**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：表达式类指针
* 获取Term
  + 函数接口：**\_Expression**\* **getTerm**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：表达式类指针
* 获取factor
  + 函数接口：**\_Expression**\* **getFactor**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：表达式类指针
* 获取else语句
  + 函数接口：**\_Statement**\* **getElseStatement**(**Token** \*now);
  + 参数列表
    - Token \*now ：当前语法分析树节点
  + 返回值：语句类指针
* 获取record的内容
  + 函数接口：void **getrecordBody**(**Token** \*now,**vector**<**\_Variant**\*> &recordList);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Variant\*> &recordList ：record列表
  + 返回值：空
* 获取for语句的操作内容
  + 函数接口：void **getforStatementOperation**(**Token** \*now,**\_ForStatement**\* &\_forStatement);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - \_ForStatement\* &\_forStatement ：for语句指针
  + 返回值：空
* 获取case语句体
  + 函数接口：void **getCaseBody**(**Token** \*now,**\_CaseStatement**\* &\_caseStatement);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - \_CaseStatement\* &\_caseStatement ：case语句指针
  + 返回值：空
* 获取case分支语句列表
  + 函数接口：

void **getBranchlist**(**Token** \*now,**\_CaseStatement**\* &\_caseStatement);

* + 参数列表
    - Token \*now ：当前语法分析树节点
    - \_CaseStatement\* &\_caseStatement ：case语句指针
  + 返回值：空
* 获取case分支语句
  + 函数接口：void **getBranch**(**Token** \*now,**\_CaseStatement**\* &\_caseStatement);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - \_CaseStatement\* &\_caseStatement ：case语句指针
  + 返回值：空
* 获取case语句中的选项常量
  + 函数接口：void **getConstlist**(**Token** \*now,**vector**<**\_Constant** \*> &constlist);
  + 参数列表
    - Token \*now ：当前语法分析树节点
    - vector<\_Constant \*>& constlist ：用于存储选项常量的数组
  + 返回值：空

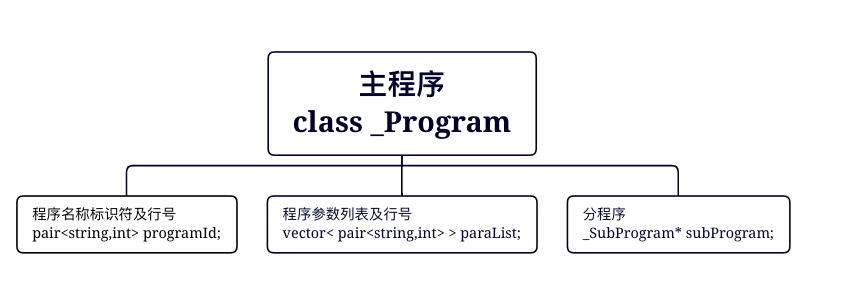
### 符号表操作接口

* 添加传值参数
  + 函数接口：void addPara(string id, int lineNumber, string type);
  + 参数列表：
    - string id 传值参数标志符
    - int lineNumber 行号
    - string type 传值参数类型
  + 返回值：无
* 添加引用参数
  + 函数接口：void addVarPara(string id, int lineNumber, string type);
  + 参数列表：
    - string id 引用参数标志符
    - int lineNumber 行号
    - string type 引用参数类型
  + 返回值：无
* 添加普通变量
  + 函数接口：void addVar(string id, int lineNumber, string type);
  + 参数列表
    - string id 普通变量标志符
    - int lineNumber 行号
    - string type 普通变量类型
  + 返回值：无
* 添加常量
  + 函数接口：void addConst(string id, int lineNumber, string type, bool isMinusShow, string value);
  + 参数列表
    - string id 常量标志符
    - int lineNumber 行号
    - string type 常量类型
    - bool isMinusShow 标志常量是否为负数
    - string value 常量取值
  + 返回值：无
* 添加数组
  + 函数接口：void addArray(string id, int lineNumber, string type, int amount, vector<pair<int, int> > arrayRangeList);
  + 参数列表
    - string id 数组类型标志符
    - int lineNumber 行号
    - string type 数组类型
    - int amount 数组维数
    - vector<pair<int, int> > arrayRangeList 数组各维上下界
  + 返回值：无
* 添加过程
  + 函数接口：void addProcedure(string id, int lineNumber, int amoun, vector<\_FormalParameter \*> paras);
  + 参数列表
    - string id 过程标志符
    - int lineNumber 行号
    - int amount 过程参数个数
    - vector<\_FormalParameter \*> paras 过程参数列表
  + 返回值：无
* 添加函数
  + 函数接口: void addFunction(string id, int lineNumber, string type, int amount,vector<\_FormalParameter\*> paras);
  + 参数列表
    - string id 函数标志符
    - int lineNumber 行号
    - string type 函数类型
    - int amount 函数参数个数
  + 返回值：无
* 添加主程序参数
  + 函数接口：void addVoidPara(string id, int lineNumber);
  + 参数列表
    - string id 主程序参数标志符
    - int lineNumber 行号
  + 返回值：无
* 添加record
  + 函数接口：void addRecords(string id, int lineNumber, vector<\_SymbolRecord \*> records);
  + 参数列表
    - string id record类型标志符
    - int lineNumber 行号
    - vector<\_SymbolRecord \*> records record类型里面的声明记录
  + 返回值：无
* 添加主程序
  + 函数接口：void addProgram(string id, int lineNumber, int amount, string returnType);
  + 参数列表
    - string id 主程序标志符
    - int lineNumber 行号
    - int amount 主程序参数个数
    - string returnType 主程序返回类型

## 各部分功能描述

### 语法分析树转AST

抽象语法树AST中主程序对应的结点如下：



入口函数：**\_Program** \*ASTRoot = **getProgram**(ParseTreeHead);

通过调用getProgram函数，将语法分析中获得的语法分析树转换成一棵抽象语法树；其中，传入的参数ParseTreeHead即语法分析树的根结点；

### 符号表的建立

符号表要求记录内容：

* 种类标志
* 标识符名字
* 行号
* 类型
* 常量取值
* 常量是否为负标志
* 参数个数/数组维数
* 数组各维上下界
* records的记录集合
* 函数过程参数

**种类标志**：记录着标识符的符号种类

* "value parameter"表示传值参数
* "var parameter"表示传引用参数
* "normal variant"表示普通变量或数组类型名或record类型名
* "constant"表示常量
* "array"表示数组
* "procedure"表示过程
* "function"表示函数
* "record"表示record集
* "program"表示主程序
* "parameter of program"表示主程序的参数

**标识符名字**：作为语义分析部分识别标识符的主键，在进行添加、查找、修改等操作时发挥重要作用

**行号**：语义分析过程中，记录下每个符号的行号，以便报错包含位置信息

**类型**：对于变量和常量来说，该域记录着变量或常量类型； 对于数组类型来说，该域记录着数组元素的类型；对于函数来说，该域记录着函数返回值类型。

**常量取值**：需要将常量的取值保存下来，以便后续计算常量表达式的值、进行常量定义时的常数传播、检查除0错误、检查数组下标越界

**常量是否为负标志**：标记常量是否为负数

**参数个数/数组维数**：在参数个数/数组维度部分，对于数组类型的变量，我们将存储其维数；对于函数或者过程类型，我们将存储其参数个数。

**数组各位上下界**：对于数组而言，在符号表中记录其各维上下界，便于判断其是否越界

**records的记录集合**：记录records集合中的各个声明语句，用于在record类型中遍历其成员

**函数过程参数**：存储函数或过程的参数集合，用于实现参数类型判断和引用等

### 语义分析及类型检查

* 对主程序进行语义分析

void **SemanticAnalyseProgram**(**\_Program** \*program);

该函数实现了对program\_head的语义分析，参数program是抽象语法树的根结点；

并完成了以下功能：

将主程序名以及主程序参数加入符号表；

将read、write等库函数加入符号表；

对program的子结点，即主程序体subProgram进行语义分析；

* 对主程序体进行语义分析

void **SemanticAnalyseSubprogram**(**\_SubProgram** \*subprogram);

在此函数中，通过不同的函数调用，对主程序中的常量定义、自定义类型定义、变量定义语句，子程序定义语句，以及复合语句进行了语义分析；

* 对常量定义语句进行语义分析

void **SemanticAnalyseConst**(**\_Constant** \*constant);

对常量语义分析的过程中，首先检查该常量是否与库函数或者主程序同名，其次会检查是否重定义；之后检查常量的类型，如果是integer，检查是否越界，如果发生越界，则将该常量类型改为real；

* 对type定义语句进行语义分析

void **SemanticAnalyseTypedef**(**\_TypeDef** \*typedefi);

对type定义进行语义分析时，首先检查该自定义类型的名称是否与库函数重复，其次检查是否重定义；

检查完毕后，根据自定义变量的类型，调用不同的语义分析函数；

* 对其他语句statement进行语义分析

void **SemanticAnalyseStatement**(**\_Statement** \*statement);

在抽象语法树结构中，\_Statement作为一个基类，许多类型的语句都继承于\_Statement，在此函数中，根据\_Statement的type字段，得到语句的类型，并做出相应的语义分析动作：

* + 复合语句\_Compound：对复合语句中的每一个基本语句，调用此函数逐一进行语义分析；
  + 赋值语句\_AssignStatement：对于赋值语句，对左值和右值进行类型检查，当左右类型不一致时，只支持integer到real的转换，否则报左右类型不一致的错误；同时需要注意的错误是，为一个常量赋值；
  + 过程调用语句\_ProcedureCall：首先根据过程的名字，查符号表，如果在符号表中查不到该过程，那么说明有过程未定义的错误；其次，如果在符号表中查到了该名称，但对应的类型并不是procedure，那么说明错把其他类型当做了过程调用；

之后，根据符号表，获取该过程的参数列表，检查procedure调用时实参的个数和类型是否和形参一致；

同时，形参又分为引用参数和传值参数，对于引用参数，我们要求首先对应的形参不能是复杂表达式、常量等，其次要求实参与形参的类型强一致，即不允许任何的隐式类型转换；而对于传值参数，只允许integer到real的隐式转换；

其中要特别注意两个库函数read和write，首先这两个库函数都是变参过程，实参个数至少为1；其次read只能读取基本类型的变量，而不能读取数组等复杂类型；

* + repeat语句\_RepeatStatement、while语句\_WhileStatement：主要检查语句中条件表达式的类型是否是boolean类型，再对循环体中的语句逐一分析；
  + if语句\_IfStatement：检查语句中条件表达式的类型是否是boolean，再对then和else之后的语句进行语义分析；
  + for语句\_ForStatement：主要对循环变量进行类型检查，首先检查该循环变量是否已经定义，其次检查是否是常量或其他无法被循环赋值的量，之后检查循环变量以及循环初值和终值的类型是否是integer；最后对循环体语句依次进行语义分析；
  + case语句\_CaseStatement：
* 对变量定义进行语义分析

void SemanticAnalyseVariant(\_Variant \*variant);

首先检查变量名是否与库函数同名，其次检查该变量在当前作用域内是否重定义，然后检查变量的类型是否是标准类型或已经定义过的类型，类型存在则加入符号表，如果变量的类型为结构体或数组并且之前没有过类型定义，则调用相应的语义分析函数进行分析，反之则提示错误信息。

* 对子程序定义进行语义分析

void SemanticAnalyseSubprogramDefinition(\_FunctionDefinition \*functionDefinition);

更新程序层数后，首先检查是否重定义，之后根据是否有返回值来确定是procrdure还是functioin，然后将记录添加到符号表中。再对形式参数进行语义分析，调用相应的语义分析函数，并将形参加入符号表中。之后依次对其中的常量定义、类型定义、变量定义、子程序（限制了程序的最大嵌套层数）和语句进行语义分析，分析完成后层数更新。

* 对形式参数进行语义分析

void SemanticAnalyseFormalParameter(\_FormalParameter \*formalParameter);

首先检查形式参数是否和库函数同名，然后检查形参是否与已经定义过的函数、过程同名，然后根据形参是传值调用还是传引用调用来调用不同的语义分析函数。

* 对函数调用进行语义分析

string SemanticAnalyseFunctionCall(\_FunctionCall \*functionCall);

首先查符号表检查函数是否已经声明，然后检查函数调用时的参数数目是否与函数声明相同，再逐个检查参数的类型是否与函数声明时一致（仅支持integer到real的隐式转换），返回函数的返回值类型。

* 对表达式进行语义分析

string SemanticAnalyseExpression(\_Expression \*expression);

根据表达式类型对表达式采取不同的处理（在表达式类型为整数是记录值是为了进行数组下标越界检查、除0检查等静态类型检查部分）：

* + 如果表达式为变量var，则进一步记录整数常量的值，表达式类型即为变量自己的类型；
  + 如果表达式为整数integer，则记录整数值，表达式类型记录为整型；
  + 如果表达式类型为其他基本类型(real,char,boolean)，则表达式类型记录为相应的基本类型；
  + 如果表达式为函数调用function则将表达式类型记录为函数返回值类型；
  + 如果表达式为复合表达式compund，则进一步根据其中运算符来对表达式成分进行类型检查：
* 如果表达式中的运算符为关系运算符relop，则检查运算符两侧的操作数表达式类型是否为integr或real类型，若不是则报错，表达式类型记录为error，否则表达式类型记录为boolean；
* 如果表达式中的运算符为单目运算符not，则检查not后的操作数表达式类型是否为boolean类型，不是则报错，表达式类型记录为error，反之表达式类型记录为boolean；
* 如果表达式中的运算符为单目运算符正号add、负号minus，则检查正号（负号）后的操作数类型是否为integer或real，若不是则报错，表达式类型想记录为error，其中integer类型需要记录整数值，最后表达式类型相应地记录为integer或real；
* 如果表达式中的运算符为算术运算符，则首先对除法操作（/,mod,div）检查是否有除0错误，然后检查算术运算符两侧操作数表达式类型是否为Integer 或 real，并且支持integer到real的隐式转换，即算术表达式中操作数既有real类型又有integer类型时，表达式类型记录为real。
* 如果表达式中的运算符为逻辑运算符（and ，or），则检查逻辑运算符两侧操作数表达式类型是否为boolean，若不是则报错，表达式类型记录为error，反之表达式类型记录为boolean。
* 对变量引用进行语义分析

string SemanticAnalyseVariantReference(\_VariantReference \*variantReference);

首先检查变量是否未定义；然后根据变量类型进行语义分析：

* 当变量不是数组元素也不是结构体属性：
* 如果该引用为函数名，则先检查该函数调用是作为赋值语句的左值还是右值，如果作为左值，则是作为当前函数的返回语句部分，那么该函数名必须是当前函数名，变量引用类型记录为funtion return reference，否则报错，变量引用类型记录为error；如果作为右值，则参数个数必须为0，因为参数个数大于0的函数名会被识别为函数调用而不是变量引用，正确情况下此时变量引用类型记录为函数返回值类型，反之记录为error；
* 其余类型只可能是传值参数，传引用参数，普通变量和常量，此时需要进行类型检查，检查类型是否为基本类型，如果不是基本类型则需要对类型别名作替换。因为非数组、非结构体的数据类型只可能是基本类型，为了后续表达式类型检查，此时需要进行替换。变量引用类型记录为相应的基本类型。
* 当变量为数组元素，首先检查变量引用的维度是否与数组定义时一致，然后逐个检查该数组元素下标表达式是否为整型，下标值（调用表达式语义分析时记录）是否越界。最后变量引用类型记录为数组元素的类型。
* 当变量为结构体属性时，首先检查使用的属性是否在结构体中存在，如果不存在则报错，否则将变量引用类型记录为结构体属性的类型。
* 当变量中数组与结构体互相嵌套（结构体的属性类型为数组、结构体的属性类型为结构体，结构体数组等，不限制嵌套层数）时，逐层构造单独的变量引用，进行类型替换，递归调用变量引用分析函数进行分析，获取最终的类型。

## 数据结构说明

### 抽象语法树AST设计

#### 1.1 结点类型说明

* 主程序

class **\_Program** *//程序(相当于program\_head)*

{

public:

**pair**<**string**, int> programId; *// PASCAL程序名称标识符及行号*

**vector**<**pair**<**string**, int> > paraList; *// PASCAL程序参数列表及行号*

**\_SubProgram** \*subProgram;*//分程序*

public:

**\_Program**();

**~\_Program**();

};

* 分程序

class **\_SubProgram** *//分程序(相当于program\_body)*

{

public:

**vector**<**\_Constant** \*> constList; *//常数定义列表*

**vector**<**\_Variant** \*> variantList; *//变量定义列表*

**vector**<**\_TypeDef** \*> typedefList; *//重构定义列表*

**vector**<**\_FunctionDefinition** \*> subprogramDefinitionList; *//过程和函数定义列表*

**\_Compound** \*compound; *//主程序体*

public:

**\_SubProgram**();

**~\_SubProgram**();

};

* 常量定义

class **\_Constant** *//常量定义*

{ *//常量定义的时候，右值可以是已经定义好的常量标识符*

*//在代码生成的时候，可以根据常量的范围对类型进行进一步的细化*

public:

**pair**<**string**, int> constId;

**string** type; *//常数类型,分为"boolean","integer","real","char"*

**pair**<**string**, int> valueId;

    char charValue;

    int intValue;

    float realValue;

    bool boolvalue;

**string** strOfVal; *//所有常量取值的字符串表示*

    bool isMinusShow; *//是否为负数*

public:

**\_Constant**() {}

**~\_Constant**() {}

};

* 变量定义

class **\_Variant** *//变量定义*

{

public:

**pair**<**string**, int> variantId; *//变量标识符ID及行号*

**\_Type** \*type; *//变量类型*

public:

**\_Variant**();

**\_Variant**(**pair**<**string**, int> \_variantId, **\_Type** \*\_type);

**~\_Variant**();

};

* 类型定义

class **\_Type** *//类型*

{

public:

**pair**<**string**, int> type; *//基本类型及行号 "integer"、"char"、"real"、"boolean"*

    int flag; *// 0表示非数组，1表示数组*

**vector**<**pair**<int, int>> arrayRangeList; *// flag=1时，表示数组各维上下界*

**vector**<**\_Variant** \*> recordList;

public:

**\_Type**();

**\_Type**(**pair**<**string**, int> \_type, int \_flag, **vector**<**pair**<int, int>> \_arrayRangeList);

**~\_Type**() {}

};

* 重构定义

class **\_TypeDef** *//重构定义*

{

public:

**pair**<**string**, int> typedefId; *//变量标识符ID及行号*

**\_Type** \*type; *//变量类型*

public:

**\_TypeDef**();

**\_TypeDef**(**pair**<**string**, int> \_typeDefId, **\_Type** \*\_type);

**~\_TypeDef**();

};

* 过程或函数定义

class **\_FunctionDefinition**

{ *//函数/过程定义*

public:

**pair**<**string**, int> functionID; *//函数/过程标识符及行号*

**vector**<**\_FormalParameter** \*> formalParaList; *//形式参数列表*

**pair**<**string**, int> type; *//如果type.first是空串，则为过程，否则为函数,取值为"integer","real","boolean","char"四种*

**vector**<**\_Constant** \*> constList; *//常数定义列表*

**vector**<**\_TypeDef** \*> typedefList; *//重构定义列表*

**vector**<**\_Variant** \*> variantList; *//变量定义列表*

**vector**<**\_FunctionDefinition** \*> subprogramDefinitionList; *//子程序和子函数定义列表*

**\_Compound** \*compound; *//主程序体*

public:

**\_FunctionDefinition**();

**~\_FunctionDefinition**();

};

* 形参定义

class **\_FormalParameter** *//形式参数*

{

public:

**pair**<**string**, int> paraId; *//形式参数标识符和行号*

**string** type; *//形式参数类型*

    int flag; *// flag=0表示传值调用，flag=1表示引用调用*

public:

**\_FormalParameter**();

**\_FormalParameter**(**pair**<**string**, int> \_paraId, **string** \_type, int \_flag);

**~\_FormalParameter**() {}

};

* 基本语句定义

class **\_Statement**

{

public:

**string** type; *//"compound","repeat","while","for","if","assign","procedure"*

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

    int lineNo; *//行号*

    bool isReturnStatement; *//是否是返回值语句*

public:

**\_Statement**() {}

**~\_Statement**() {}

};

* 复合语句定义

class **\_Compound** : public **\_Statement**

{

public:

**vector**<**\_Statement** \*> statementList; *//语句列表*

*//行号由begin的位置决定*

public:

**\_Compound**();

**~\_Compound**();

};

* 赋值语句

class **\_AssignStatement** : public **\_Statement**

{

public:

**\_VariantReference** \*variantReference; *//左值变量*

**\_Expression** \*expression; *//右值表达式*

*//行号由赋值符号的位置决定*

public:

**\_AssignStatement**();

**~\_AssignStatement**();

};

* 过程调用语句

class **\_ProcedureCall** : public **\_Statement**

{

public:

**pair**<**string**, int> procedureId; *//过程标识符*

**vector**<**\_Expression** \*> actualParaList; *//实际参数列表，由表达式组成*

*//行号由procedure名的位置决定*

public:

**\_ProcedureCall**();

**~\_ProcedureCall**();

};

* 函数调用语句

class **\_FunctionCall**

{

public:

**pair**<**string**, int> functionId; *//函数标识符*

**vector**<**\_Expression** \*> actualParaList; *//实际参数列表，由表达式组成*

**string** returnType; *//"integer"、"real"、"char"、"boolean"、"error"，其中error表示函数标识符不存在*

public:

**\_FunctionCall**();

**~\_FunctionCall**();

};

* 表达式定义

class **\_Expression**

{

public:

**string** type; *//表达式类型,"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*

**\_VariantReference** \*variantReference; *//变量或常量或数组*

    int intNum; *//整数*

    float realNum; *//浮点数*

**string** strOfNum; *//整数和浮点数的string表示（考虑从PASCAL-S源程序将字符串转为浮点数，再将浮点数转为字符串会带来精度问题，所以需要存下初始字符串值）*

    char charVal; *//常量字符*

**\_FunctionCall** \*functionCall; *//函数调用*

    int isMinusShow; *//是否为负*

**string** boolValue; *//bool类型记录值，值为“true”或false*

**string** operation; *//具体操作符*

**string** operationType; *//操作符类型,"relop","mulop","addop","single"*

**\_Expression** \*operand1, \*operand2;

    int lineNo; *//行号, 用表达式中最先出现的操作数的行号表示*

public:

**\_Expression**();

**~\_Expression**();

*//语义分析相关*

public:

    int totalIntValue;

    bool totalIntValueValid;

**string** expressionType; *//区别于type，这个值表示表达式的具体类型，即"integer"、"real"、"char"、"boolean"、"error"，其中error表示表达式中包含类型不一致的操作数*

};

* 表达式中变量引用定义

class **\_VariantReference**

{

public:

**pair**<**string**, int> variantId; *//变量或常量标识符和行号*

*//如果这个变量是结构体或数组：*

**vector**<**\_Idvpart** \*> IdvpartList; *//结构体.属性或数组元素*

*//        int flag;*

*//        string str;*

public:

**\_VariantReference**();

**~\_VariantReference**();

public:

    int locFlag; *//-1表示左值，1表示右值，0表示什么都不是 左值特判*

**string** kind; *//"array","var","constant","function call","function return reference"*

**string** variantType; *//"integer"、"real"、"char"、"boolean"、"error"，其中"error"表示数组某一维下标表达式的类型不为"integer"或标识符不存在*

};

* 变量后缀定义

class **\_Idvpart**

{

public:

**vector**<**\_Expression** \*> expressionList; *// flag = 0;*

**pair**<**string**, int> IdvpartId; *// flag = 1*

    int flag; *// flag=0表示数组*

public:

**\_Idvpart**();

**~\_Idvpart**();

};

* if语句

class **\_IfStatement** : public **\_Statement**

{

public:

**\_Expression** \*condition; *//条件表达式*

**\_Statement** \*then; *//满足条件时执行的语句*

**\_Statement** \*els; *//不满足条件时执行的语句，如果为NULL，则没有else部分*

*//行号由if的位置决定*

public:

**\_IfStatement**();

**~\_IfStatement**();

};

* case语句

class **\_CaseStatement** : public **\_Statement**

{

public:

**\_Expression** \*caseid;

**vector**<**\_Branch** \*> branch;

**vector**<**\_Statement** \*> \_do;

**\_CaseStatement**();

**~\_CaseStatement**();

};

* case分支语句

class **\_Branch**

{

public:

**vector**<**\_Expression** \*> condition; *//分支条件*

**\_Statement** \*\_do; *//执行的语句体*

**\_Branch**();

**~\_Branch**();

};

* for语句

class **\_ForStatement** : public **\_Statement**

{

public:

**pair**<**string**, int> id; *//循环变量*

**\_Expression** \*start; *//起始值*

**\_Expression** \*end; *//终止值*

**\_Statement** \*\_do; *//循环体语句*

*//行号由for的位置决定*

public:

**\_Expression** \*condition; *//判断条件*

**\_AssignStatement** \*increment; *//增量*

**\_AssignStatement** \*initial; *//初始化*

**\_ForStatement**();

**~\_ForStatement**();

};

* repeat语句

class **\_RepeatStatement** : public **\_Statement**

{

public:

**\_Expression** \*condition; *//条件表达式*

**vector**<**\_Statement** \*> \_do; *//循环体语句*

*//行号由repeat的位置决定*

public:

**\_RepeatStatement**();

**~\_RepeatStatement**();

};

* while语句

class **\_WhileStatement** : public **\_Statement**

{

public:

**\_Expression** \*condition; *//条件表达式*

**\_Statement** \*\_do; *//循环体语句*

*//行号由while的位置决定*

public:

**\_WhileStatement**();

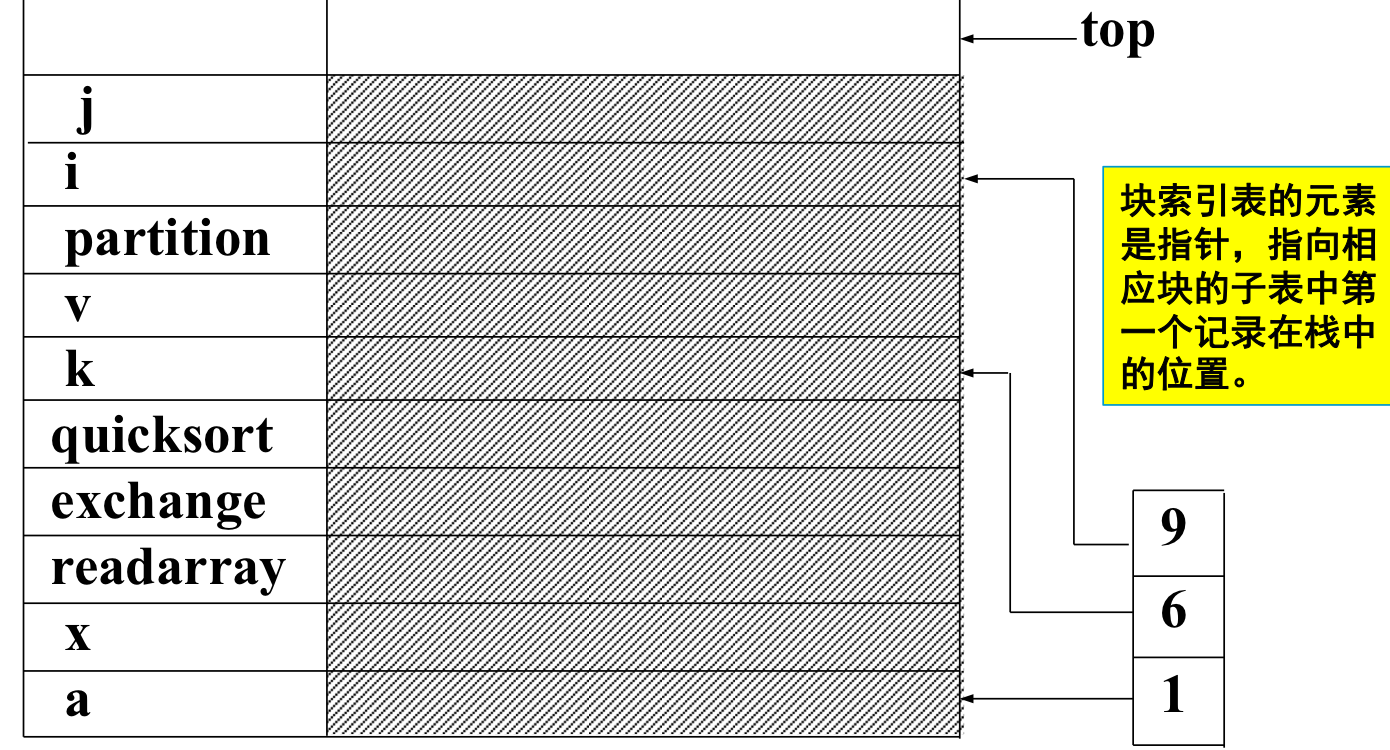
**~\_WhileStatement**();

};

### 符号表的设计

符号表设计为栈式符号表

* 当遇到变量声明时，将包含变量属性的记录入栈
* 当到达块结尾时，将该块中声明的所有变量的记录出栈，因为这些变量是局部于该块的，在块外已不可用



#### 2.1 关键数据类型

* recordList：栈式存储符号表的记录
* indexTable：表示每个函数过程块在符号表中开始索引位置的块索引表 即indexTable[i]=j : 过程i的记录在符号表中第j个位置
* idToLoc：表示每个记录在符号表的位置，用map存储方便查找
* custom：存储自定义类型的名字

同时，将符号表项和符号表分为两个类进行存储，做到记录和总表的隔离性。符号表项记录类为\_SymbolRecord；符号表类为\_SymbolTable。在类\_SymbolRecord中，通过set函数来这是记录属性，通过\_SymbolTable类中的add函数进行符号表的添加。

#### 2.2 不同类型符号表的存储情况

* 常量类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | constant |
| id | 常量名字 |
| lineNumber | 常量所在行号 |
| type | 常量类型 |
| value | 常量值 |
| isMinusShow | 标记该常量是否为负数 |

* 变量类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | normal variant |
| id | 变量名字 |
| lineNumber | 变量所在行号 |
| type | 变量类型 |

* 数组类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | array |
| id | 数组类型名字 |
| lineNumber | 数组所在行号 |
| type | 数组类型 |
| amount | 数组维数 |
| arrayRangeList | 数组各维上下界 |

将数组类型名字存储到custom中

* record类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | record |
| id | record类型名字 |
| lineNumber | record声明所在行号 |
| records | record下所属符号表记录 |

将record类型名字存储到custom中

* 自定义类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | normal variant |
| id | 自定义类型名 |
| lineNumber | 自定义类型所在行号 |
| type | 自定义类型 |

将自定义类型名字存储到custom中

* 函数类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | function |
| id | 函数名字 |
| lineNumber | 函数声明所在行号 |
| type | 函数返回类型 |
| amount | 函数参数个数 |
| paras | 函数参数列表 |

* 过程类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | procedure |
| id | 过程名字 |
| lineNumber | 过程所在行号 |
| amount | 过程参数个数 |
| paras | 过程参数列表 |

* 参数类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | "value parameter"表示传值参数  "var parameter"表示传引用参数 |
| id | 参数名字 |
| lineNumber | 参数所在行号 |
| type | 参数类型 |

* program类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | program |
| id | program名字 |
| lineNumber | program声明所在行号 |
| type | program返回类型 |
| amount | program参数个数 |

* program 参数类型

|  |  |
| --- | --- |
| 属性名 | 属性值 |
| flag | parameter of program |
| id | program参数名字 |
| lineNumber | program参数所在行号 |

注意：对于array、record类型，这两种类型在声明时在符号表中会先创建array、record类型，再进行变量声明。对于没有给出具体类型名的情况会通过变量名后加下划线"\_"的方式进行类型命名。

## 算法描述

* 在符号表中查找对应记录

**\_SymbolRecord** \***findSymbolRecord**(**string** id);

通过符号表中的快速索引表idToLoc，就可以根据标识符的名称，快速定位到该标识符在符号表中最近一次的索引，从而获得对应索引；

* 检查重定义

bool **isReDef**(**string** id, int& loc);

重定义要结合变量的作用域进行检测，所以在符号表中查找对应记录的基础上，结合过程索引表indexTable，确认查找到的记录是否位于当前标识符的作用域内，从而检查是否重定义；

* 符号表重定位

void **relocation**();

当一个function或者procedure的定义结束之后，需要将对应的栈弹出，以实现变量的作用域定义，此时采取的做法是：根据indexTable找到当前作用域开始处的索引，对从此开始的每一条索引，都取出其名称，再根据名称从inToLoc索引表和custom索引表出栈；

* 提取表达式和变量引用的内容

void **inputExpression**(**\_Expression**\* expression, **string**& ans);

void **inputVariantRef**(**\_VariantReference** \*leftVariantReference, **string**& ans);

当某个表达式出现类型错误之后，在报错信息中需要输出该表达式的内容；由于表达式内容均以多叉树的结构存储，在此通过两个函数，遍历表达式\_Expression结点和变量引用\_VariantReference结点所在的树，根据结点类型的不同作不同的遍历方法，即可输出表达式和变量引用的内容。