# 目录

[1 目录 1](#_Toc513585250)

[2 代码生成详细设计 2](#_Toc513585251)

[2.1 术语说明 2](#_Toc513585252)

[2.2 代码生成方法 3](#_Toc513585253)

[2.2.1 代码生成概述 3](#_Toc513585254)

[2.2.2 对应关系分析 3](#_Toc513585255)

[2.2.2.1 头文件 3](#_Toc513585256)

[2.2.2.2 程序结构 4](#_Toc513585257)

[2.2.2.3 类型关键字的对应关系 4](#_Toc513585258)

[2.2.2.4 运算符的对应关系 4](#_Toc513585259)

[2.2.2.5 引用参数与指针 5](#_Toc513585260)

[2.2.2.6 常量和变量定义 5](#_Toc513585261)

[2.2.2.7 主程序参数列表 5](#_Toc513585262)

[2.2.2.8 数组下标 5](#_Toc513585263)

[2.2.2.9 数组引用列表 5](#_Toc513585264)

[2.2.2.10 返回语句 5](#_Toc513585265)

[2.2.2.11 复合语句块 6](#_Toc513585266)

[2.2.2.12 循环语句 6](#_Toc513585267)

[2.2.2.13 if分支语句 6](#_Toc513585268)

[2.2.2.14 赋值语句 6](#_Toc513585269)

[2.2.2.15 程序的无参调用 6](#_Toc513585270)

[2.2.2.16 输入调用 6](#_Toc513585271)

[2.2.2.17 输出调用 7](#_Toc513585272)

[2.2.2.18 程序头 7](#_Toc513585273)

[2.2.2.19 程序参数列表 7](#_Toc513585274)

[2.2.3 重要细节 7](#_Toc513585275)

[2.2.3.1 头文件标记方法 7](#_Toc513585276)

[2.2.3.2 bool表达式的输出 8](#_Toc513585277)

[2.2.3.3 数组下标转化 8](#_Toc513585278)

[2.2.3.4 语句缩进控制 8](#_Toc513585279)

[2.2.3.5 表达式优先级和输出时的括号 9](#_Toc513585280)

[2.2.3.6 引用参数转化为指针 10](#_Toc513585281)

[2.2.3.7 程序头中的引用参数 10](#_Toc513585282)

[2.2.4 生成的C程序结构 10](#_Toc513585283)

[2.3 代码生成涉及的数据结构 11](#_Toc513585284)

[2.3.1 头文件 11](#_Toc513585285)

[2.3.2 全局常量 11](#_Toc513585286)

[2.3.3 全局变量 12](#_Toc513585287)

[2.3.4 原PASCAL-S主程序对应到C程序的接口声明 12](#_Toc513585288)

[2.3.5 子过程/子函数接口声明 12](#_Toc513585289)

[2.3.6 主程序语句列表 12](#_Toc513585290)

[2.3.7 子程序 12](#_Toc513585291)

[2.3.8 符号表指针 13](#_Toc513585292)

[2.4 引用的外部函数接口 13](#_Toc513585293)

[2.5 获取"代码数据结构的函数设计 14](#_Toc513585294)

[2.5.1 获取头文件 14](#_Toc513585295)

[2.5.2 获取常量列表 14](#_Toc513585296)

[2.5.3 获取变量列表 15](#_Toc513585297)

[2.5.4 获取子程序接口声明列表 16](#_Toc513585298)

[2.5.5 获取函数调用 16](#_Toc513585299)

[2.5.6 获取表达式 17](#_Toc513585300)

[2.5.7 获取变量引用 20](#_Toc513585301)

[2.5.8 获取语句列表 21](#_Toc513585302)

[2.5.9 获取子程序定义 23](#_Toc513585303)

[2.5.10 获取子程序体列表 24](#_Toc513585304)

[2.5.11 获取原PASCAL主程序头对应到的C程序头及程序体 24](#_Toc513585305)

[2.5.12 根据类型获取输入输出格式控制符 25](#_Toc513585306)

[2.5.13 检查并获取库程序 25](#_Toc513585307)

[2.6 输出代码的函数设计 27](#_Toc513585308)

[2.6.1 输出头文件 27](#_Toc513585309)

[2.6.2 输出常数列表 28](#_Toc513585310)

[2.6.3 输出变量列表 28](#_Toc513585311)

[2.6.4 输出子程序接口声明 29](#_Toc513585312)

[2.6.5 输出子程序接口声明列表 30](#_Toc513585313)

[2.6.6 输出语句 30](#_Toc513585314)

[2.6.7 输出语句列表 31](#_Toc513585315)

[2.6.8 输出子程序定义列表 31](#_Toc513585316)

[2.6.9 输出main函数体 32](#_Toc513585317)

[2.7 其它函数设计 32](#_Toc513585318)

[2.7.1 代码生成之前的初始化 32](#_Toc513585319)

[2.7.2 判断数字是否相等（变参） 33](#_Toc513585320)

[2.7.3 将PASCAL-S的类型关键字转化为C的类型关键字 33](#_Toc513585321)

[2.7.4 将PASCAL-S的运算符转化为C的运算符 34](#_Toc513585322)

[2.7.5 代码生成对外接口 35](#_Toc513585323)

[2.7.6 提供给语义分析的接口说明 36](#_Toc513585324)

# 代码生成详细设计

## 术语说明

|  |  |
| --- | --- |
| 术语名称 | 描述 |
| 头文件区 | 根据C程序所用到的库函数、过程，include相关头文件 |
| 全局常量定义区 | 对应PASCAL-S程序中开头的常量定义 |
| 全局变量定义区 | 对应PASCAL-S程序中开头的变量定义 |
| 子过程/子函数接口声明区 | 子过程/子函数、以及main\_function函数的接口声明区域 |
| main函数区 | 调用main\_function函数 |
| main\_function函数定义区 | PASCAL的主程序带有名称标识符，为此，C程序中专门设置一个名称为该标识符，返回值为void的函数，并在main函数中调用该函数，记该函数为main\_function，以区别于main函数，该函数内部的语句对应PASCAL-S程序的程序体 |
| 子过程/子函数定义区 | 对应PASCAL-S程序中各个子过程、子函数的定义 |
| 变量引用和引用参数 | 引用参数指的是PASCAL-S源程序中，参数被定义为引用形式；而变量引用的概念对应的是一个语法成分，即标识符或者标识符加上数组下标引用列表，变量引用可以是常量标识符、变量标识符、数组元素、无参函数调用等可以对外提供取值的成分。 |

## 代码生成方法

### 代码生成概述

代码生成的一种做法是遍历AST和调用符号表信息，直接一边遍历，一边生成代码。也可以将树形AST结构转化为**由字符串这一基本类型组成**的复杂**线性数据结构**中，再调用输出函数进行输出，后者虽工作量稍大，但代码逻辑更加清晰，便于维护和扩展，**所以本小组采用后者实现代码生成**。

①遍历AST、查符号表，将代码保存到线性数据结构，这些数据结构将在下一部分进行介绍

②按照C程序结构，遍历各线性数据结构，输出代码

### 对应关系分析

#### 头文件

头文件的需求来自于库程序的调用，PASCAL-S仅支持read、write、writeln、exit四种库程序的调用，其中exit对应了返回值语句，所以剩下的三个过程对应到C程序就是printf和scanf这两个语句，这两个语句包含在**stdio.h**头文件中。我们将在所有生成函数调用、过程调用代码的位置调用一个函数专门来检查所调用和函数和过程是否是库程序，如果是库程序，则需标记相应的头文件。最后输出代码的时候，输出被标记了的头文件即可。

这样设计而不是对read、write、writeln进行特判，保证了我们编译器的可扩展性，后续如果要添加对别的库程序的支持，对于代码的修改将会特别容易进行。

另外，在C99中C语言加入了对bool类型的支持，但仍需加入**stdbool.h**头文件，考虑到boolean类型是PASCAL-S程序的基本类型，生成的C语言代码中，将自动加入stdbool.h头文件。

**综上，如果源程序包含输入输出语句，则输出的C代码会包含stdio.h头文件，同时输出的C代码必然会包含stdbool.h头文件。**

#### 程序结构

PASCAL-S程序中不包含头文件之类的成分，如何在C程序中确定需要的头文件已在上文阐述。

PASCAL-S程序的主程序名在C语言中没有对应的内容，因为在C语言中，入口函数固定名称为main。我们需要在C程序中专门设置一个程序，取名为PASCAL程序的主程序名。然后main函数只要简单的调用该函数即可。

PASCAL-S程序中，定义在主程序部分的常量和变量可以被所有的子程序引用，这相当于C语言中的全局变量和全局常量。

结构良好的C程序需要将所有程序的声明放在main函数之前，这些程序的声明中，除了包括PASCAL-S程序中声明的所有子程序，还应包括我们在上文提到的PASCAL-S主程序对应到C程序的声明。

main函数中只需要调用PASCAL-S主程序对应到的C程序即可。

最后还应包含所有子程序的定义，当然也应包含PASCAL-S主程序对应到的C程序的定义。

#### 类型关键字的对应关系

|  |  |  |
| --- | --- | --- |
| 类型 | PASCAL-S关键字 | C关键字 |
| 整型 | integer | int |
| 浮点型 | real | float |
| 字符型 | char | char |
| 布尔型 | Boolean | bool |

#### 运算符的对应关系

|  |  |  |
| --- | --- | --- |
| 运算符 | PASCAL-S符号 | C符号 |
| 加法 | + | + |
| 减法 | - | - |
| 乘法 | \* | \* |
| 除法 | / | / |
| 整除 | div | / |
| 取余 | mod | % |
| 与 | and | && |
| 或 | or | || |
| 非 | not | ! |
| 大于 | > | > |
| 大于等于 | >= | >= |
| 小于 | < | < |
| 小于等于 | <= | <= |
| 等于 | = | == |
| 不等于 | <> | != |
| 赋值 | 常量初始化时为=  赋值语句中为:= | = |

#### 引用参数与指针

PASCAL-S支持引用参数，C不支持，所以在C中只能用指针来代替参数引用，由于我们记录类型，所以涉及的指针操作不会很复杂。

首先在调用程序时，形参变量前需加入取地址符；其次在程序定义时，需将引用参数的定义改为对应类型的指针；在程序体内，所有涉及到原引用参数的，都需要加上解引用符；还需特殊考虑引用参数又作为实参调用，同时对应的形参也是引用参数的情形，此时实参直接为该指针变量即可，不需要解引用符，当然也可以解引用后再取地址。

具体转化方法将在后文介绍。

#### 常量和变量定义

对于常量定义，pascal中的const关键字作用域较大，不局限于下一个分号，而C语言中，每一分号隔出的部分都要单独使用一个const关键字。且pascal中的常量在定义时不需要指明类型，但是C语言需要。所以在我们需要在词法分析阶段完成对常量类型的自动识别，在目标代码生成时，指明对应的类型。

pascal在声明变量时，除了要说明类型，还要再前面加上var关键字，C语言中没有这样的关键字，只需要指明类型即可。var关键字的作用域与const关键字相同。

#### 主程序参数列表

pascal主程序头中包含了一个无类型的标识符列表。经查阅资料，这个标识符列表类似于c语言中main函数的参数列表，但由于PASCAL-S缺少相应的语法支持，导致主程序参数没有任何实际用途，所以我们的编译器在代码生成阶段，将忽略主程序参数列表。

#### 数组下标

PASCAL-S的数组下标上下界均可由用户定义，实际上，上下界甚至可以是负数，但由于文法限制，PASCAL-S仅支持无符号数的数组上下界。在C程序中，数组的下界是固定的，也就是0，程序定义数组时，给出每一维的大小为ni，也就定义了每一维的上界为ni-1。下标转化方法将在后文给出。

#### 数组引用列表

PASCAL-S的数组下标引用列表用中括号包围，每一维的下标表达式之间用逗号分隔。

C的数组每一维的下标表达式均用中括号包括，同时也起到了分隔的作用。

#### 返回语句

|  |  |  |
| --- | --- | --- |
| 类型 | PASCAL-S的返回语句 | C的返回语句 |
| 函数返回值语句 | 函数名:=表达式; | return (表达式); |
| exit(表达式); | return (表达式); |
| 过程返回语句 | exit; | return; |

#### 复合语句块

PASCAL-S中复合语句块由BEGIN和END关键字包括，而C语言中则由一对大括号包括。

PASCAL-S中，复合语句块的最后一条语句后可以没有分号，但实际上按照文法来说，最后一条语句后面一定没有分号，如果从形式上看最后一条语句后面是有分号的，那么就可以认为最后一条语句为空语句（这么说非常的违反直觉，但实际上文法就是这么定义的）。所以在PASCAL-S中，即使看上去只有一条语句，如果这条语句后面出现了一个分号，那这一条语句也必须用BEGIN和END包括为一个复合语句块，if语句的then语句部分就是一个典型的例子；但是**C程序的每一条语句都必须包含分号**。

#### 循环语句

|  |  |
| --- | --- |
| PASCAL-S循环语句 | C循环语句 |
| for 循环变量 := 初值表达式 to 终值表达式 do 循环语句体 | for(循环变量 = 初值表达式 ; 循环变量 <= 终值表达式 ; 循环变量++)循环语句体 |
| while 条件表达式 do循环语句体 | while(条件表达式) do循环语句体 |
| repeat循环语句体until 条件表达式 | do循环语句体while(条件表达式) |

注意，PASCAL-S的repeat-until循环语句的含义为，执行循环语句体，直到满足条件；而C的do-while循环语句的含义为，执行循环语句体，直到条件不满足，所以在转化时，需要将在原有条件表达式的基础上取反。

#### if分支语句

在if分支语句上，两者并没有多大的差别。

PASCAL-S中的结构为： if 条件表达式 then 语句;

if 条件表达式 then 语句1 else 语句2;

C中的结构为： if(条件表达式) 语句;

if(条件表达式) 语句1 else 语句2;

#### 赋值语句

两者的区别仅在于赋值号上，PASCAL-S中为:=，C中为=，这在运算符的对应关系中已经说明。

#### 程序的无参调用

PASCAL-S中无参调用时，只需要程序名即可；C中无参调用时，需要在程序名后加上一对括号，括号中不包含任何内容。

#### 输入调用

PASCAL-S调用read进行输入，C调用scanf进行输入；

read在调用时，只需要提供用逗号分隔的普通变量（或数组元素）列表即可，而scanf不仅需要变量列表，还需一个格式控制符字符串。

类型与其格式控制符的对应关系如下：

|  |  |
| --- | --- |
| 类型 | 格式控制符 |
| int | %d |
| float | %f |
| char | %c |

PASCAL-S不支持boolean类型变量的输入。

#### 输出调用

PASCAL-S调用write或writeln进行输出，C调用printf进行输出；

write在调用时，只需要提供用逗号分隔的表达式列表即可，而printf不仅需要表达式列表，还需一个格式控制符字符串。

writeln除了需要在格式控制符字符串的最后加入一个换行符，与write没有任何区别。

其类型与格式控制符的对应关系可以参考输入。

另外还需强调输出对于boolean变量的支持。**PASCAL-S支持boolean表达式的输出**，而C程序不支持bool类型的直接输出，一种常见的做法是用%d格式控制符，这也得益于C程序对于布尔型和整型之间的隐式类型转换规则，如下表所示：

|  |  |
| --- | --- |
| 布尔型取值 | 整型取值 |
| true | 1 |
| false | 0 |

但是在PASCAL-S中，输出布尔表达式的值时，输出的就是true或者false。为了实现这种输出效果，我们并没有采用上文的解决方案，具体方法将在后文给出，这里仅讨论对应关系。

#### 程序头

|  |  |  |
| --- | --- | --- |
|  | PASCAL-S | C |
| 过程 | procedure 过程名(参数列表); | void 过程名(参数列表); |
| 函数 | function 函数名(参数列表):返回值类型; | 返回值类型 函数名(参数列表); |

可见C程序对于函数和过程的定义更加统一，可以将“void”视为一种特殊的返回值类型。

#### 程序参数列表

PASCAL-S的结构为

[var] 标识符列表 : 类型 ; [var] 标识符列表 : 类型 ; ……

C的结构为

类型 标识符, 类型 标识符, ……

PASCAL-S中引用参数需要加上var关键字，C中引用参数需要在类型后加上“&”。

### 重要细节

#### 头文件标记方法

每次碰到函数或过程调用时，该函数将检查函数或过程名是否属于库程序，如果属于库程序，需要标记其所属的头文件。

先介绍两个数据结构。

1. map<string,string> mp\_subprogramToHeadFile;//过程、函数到C程序头文件的映射
2. map<string,**bool**> mp\_headFileShow;//头文件是否出现的映射

我们首先检查mp\_subprogramToHeadFile的键值中是否包含当前程序名，如果包含，则以mp\_subprogramToHeadFile的属性值作为mp\_headFileShow的键值，将mp\_headFileShow的属性值标记为true。

#### bool表达式的输出

前面已经提到PASCAL-S中，输出boolean类型是按照原样输出的，即为真时输出true，为假时输出false，我们也将实现这种效果。

write语句的多个输出表达式，将以bool表达式进行分隔，转化为多个部分，每个bool表达式本身作为一个部分。例如write(表达式1，表达式2，表达式3，表达式4，表达式5，表达式6)，如果表达式3和表达式4是bool表达式，那么将分为以下四部分：

* 1、2
* 3
* 4
* 5、6

如果当前部分是不含bool类型的，则按照正常方法转换；

如果当前部分是一条bool表达式，则转化为如下形式：

if(bool表达式)

printf(“true”);

else

printf(“false”);

#### 数组下标转化

假设PASCAL中某一维的下标上下界为[a,b]，那么C中的上下界就为[0,b-a]。

假设引用该维的表达式为X，则在代码生成时，该表达式应写为X-a。

#### 语句缩进控制

为了保持输出代码的整洁，需要为每一条语句定义一个缩进值，即该语句前面的制表符个数。

需要控制缩进的情况如下：

|  |  |
| --- | --- |
| 情况 | 缩进 |
| 头文件 | 缩进值为0 |
| 全局常量定义 | 缩进值为0 |
| 全局变量定义 | 缩进值为0 |
| 程序声明 | 缩进值为0 |
| 程序定义的程序头 | 缩进值为0 |
| 程序定义的一对大括号 | 缩进值为0 |
| 程序定义的最外层语句 | 缩进值为1 |
| if语句 | then语句和else语句的缩进值比if和else关键字多1 |
| while语句 | 循环体语句的缩进值比其关键字所在语句的缩进值多1 |
| repeat语句 | 循环体语句的缩进值比其关键字所在语句的缩进值多1 |
| for语句 | 循环体语句的缩进值比其关键字所在语句的缩进值多1 |
| compound语句 | 每一条复合语句的缩进值比BEGIN和END关键字的缩进值多1 |

#### 表达式优先级和输出时的括号

实际上这一部分无需任何处理，因为我们的表达式抽象成的AST子树已经完美的阐述了运算符之间的优先级关系，也无需自己增添括号。

但是考虑到编译器的可扩展性，我们设计了一个根据优先级添加括号的算法，同时也可以使得一些表达式的更加易读。

例如语句a:=--b; 经过我们的处理后，生成的C代码为a=-(-b)。

首先我们为表达式赋予优先级的概念，**表达式的优先级就是其AST树形结构的根节点运算符的优先级**，可以用如下表表示：

|  |  |  |
| --- | --- | --- |
| expression->type的取值 | 运算符 | 优先级编号 |
| var | 无 | 0 |
| integer | 无 | 0 |
| real | 无 | 0 |
| char | 无 | 0 |
| function | 无 | 0 |
| compound | bracket | 4 |
| minus | 4 |
| not | 4 |
| mulop | 3 |
| addop | 2 |
| relop | 1 |

对于每一个包含运算符的复杂表达式，需要考虑其操作数表达式是否需要外加括号，其操作数表达式的优先级，也就是其操作数表达式对应的AST树形结构的根节点的运算符的优先级，如果比当前运算符优先级低，就需要给操作数表达式外加括号，具体策略如下：

|  |  |  |
| --- | --- | --- |
| expression->type的取值 | 运算符 | 添加括号的情况 |
| compound | bracket | 无，因为其本身就是添加括号 |
| minus | 操作数表达式优先级为1,2,3,4 |
| not | 操作数表达式优先级为1,2,3,4 |
| mulop | 操作数表达式优先级为1,2 |
| addop | 操作数表达式优先级为1 |
| relop | 无 |

#### 引用参数转化为指针

假设该变量为a，按照下表分情况讨论，其输出代码分别为：

|  |  |  |  |
| --- | --- | --- | --- |
| 当前变量是否是所在程序的引用参数 | 当前变量是否作为某个程序调用的实参 | 如果作为某个程序的实参，对应的形参是否是引用参数 | 输出代码 |
| × | × | × | a |
| × | × | √ | 不存在这种情况 |
| × | √ | × | a |
| × | √ | √ | &a |
| √ | × | × | \*a |
| √ | × | √ | 不存在这种情况 |
| √ | √ | × | \*a |
| √ | √ | √ | a |

对上面的表格，去除不存在项，并“合并同类项”，精简为下表：

|  |  |  |
| --- | --- | --- |
| 当前变量是否是所在程序的引用参数 | 当前变量是否作为某个程序调用的实参，且对应的形参为引用参数 | 输出代码 |
| × | × | a |
| × | √ | &a |
| √ | × | \*a |
| √ | √ | a |

所以在函数和过程调用时，需要检查实参对应的形参是否为引用参数，将该信息作为布尔型变量的参数传入到获取变量代码的函数中，然后在该函数中，再获取该变量是否是当前所在程序的引用参数，结合上表就可以得知最终应该输出怎样的代码。

#### 程序头中的引用参数

只需要将引用参数定义为对应类型的指针变量即可。

### 生成的C程序结构

C程序结构如下所示，注意main\_function在实际的代码中，应被PASCAL-S的主程序名取代。

1. //Head files
3. //Overall constant definiton
5. //Overall variable definition
7. //Subprogram declaration
8. **void** main\_function();
9. **int** fun();
10. **void** pro();
11. ……
13. //Main function
14. **int** main()
15. {
16. test();
17. **return** 0;
18. }
20. //Subprogram definition
21. **void** main\_function(){
22. ……
23. }
25. **int** fun(){
26. ……
27. }
29. **void** pro(){
30. ……
31. }
33. ……

## 代码生成涉及的数据结构

这些数据结构均为全局变量。

### 头文件

1. map<string,string> mp\_subprogramToHeadFile;//过程、函数到C程序头文件的映射
2. map<string,**bool**> mp\_headFileShow;//头文件是否出现的映射

事先将过程、函数及其对应的头文件保存到mp\_subprogramToHeadFile中；并将所有的头文件作为键值保存到mp\_headFileShow中，对应的属性值均初始化为false。

### 全局常量

1. vector<string> v\_s\_constIdList;//常量标识符
2. vector<string> v\_s\_constTypeList;//常量类型,"int","float","char","bool"
3. vector<string> v\_s\_constValueList;//常量值,不论是哪种类型，均用string表示

常量的存放顺序按照PASCAL-S程序的定义顺序，且**默认语义分析已经完成了常量定义区的常数传播**。注意常量是数值类型且是负的，则常量值字符串需要在最前面加上一个负号。

### 全局变量

1. vector<string> v\_s\_variantIdList;//变量标识符
2. vector<string> v\_s\_variantTypeList;//变量类型,"int","float","char","bool"
3. vector< vector<**int**> > v\_ArraySizeList;//各数组变量各维的大小

v\_ArrayRangeList的size可以用来指明变量是否为数组，若为数组，数组的维数是多少。

### 原PASCAL-S主程序对应到C程序的接口声明

1. string subMainFunctionDeclaration;//原PASCAL主程序对应到C程序的声明

该接口声明单独拿出来，区别于其它子过程/子函数的接口声明。

### 子过程/子函数接口声明

1. **struct** subproDef{
2. string returnType;//子程序返回值类型,"int","float","char","bool"
3. string id;//子程序名称标识符
4. vector<string> v\_paraIdList;//参数标识符
5. vector<**bool**> v\_isParaRef;//参数是否为引用
6. vector<string> v\_paraTypeList;//参数类型,"int","float","char","bool"
7. };
8. vector< **struct** subproDef > v\_subproDefList;//子程序接口列表

子程序接口列表中的第一个子程序永远是main\_function

### 主程序语句列表

1. vector< pair<string,**int**> > v\_statementList;

每一个pair的first表示语句本身，second表示语句的缩进，即语句前的制表符个数。

### 子程序

1. //子程序定义
2. **struct** subproDef{
3. vector<string> constIdList;//常量标识符
4. vector<string> constTypeList;//常量类型,"int","float","char","bool"
5. vector<string> constValueList;//常量值,不论是哪种类型，均用string表示
7. vector<string> variantIdList;//变量标识符
8. vector<string> variantTypeList;//变量类型,"int","float","char","bool"
9. vector< vector<**int**> > arraySizeList;//各数组变量各维的大小
11. vector< pair<string,**int**> > statementList;//pair的first是语句本身，pair的second控制缩进，即语句前的制表符个数
12. };
13. //子程序定义列表
14. vector< **struct** subproDef > subproDefList;

子程序定义中，常量定义、变量定义与全局常量定义和全局变量定义类似；statementList是子程序的语句列表，pari的first是语句本身，pair的second控制缩进，即语句前的制表符个数。

### 符号表指针

1. **extern** \_SymbolTable \*mainSymbolTable;
2. \_SymbolTable\* codeGenerateCurrentSymbolTable;

其中mainSymbolTable是主符号表指针，由语义分析程序提供；而codeGenerateCurrentSymbolTable是用于代码生成时表示当前符号表的指针，用于符号表的定位和重定位。

## 引用的外部函数接口

只用到了符号表查找记录的函数。

* 函数接口

1. \_SymbolRecord\* findSymbolRecord(\_SymbolTable\* currentSymbolTable, string id, **int** mode = 0);

* 函数功能

从指定符号表中，以给定的标识符，查找其记录指针

* 返回值

\_SymbolRecord\*，表示查找到的记录指针，如果没有找到，则为NULL

* 参数

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_SymbolTable\* currentSymbolTable | 指定的符号表指针 |
| string id | 指定的标识符 |
| **int** mode | 如果mode=0，表示在内层符号表没有找到时，需要到外层符号表查找；如果mode!=0，表示无需到外层符号表查找 |

## 获取"代码数据结构的函数设计

包括函数接口和伪代码，会用到5中定义的数据结构和抽象语法树的节点类。

### 获取头文件

* 函数接口

1. **void** inputHeadFileList(string subprogramId);

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** inputHeadFileList(string subprogramId){
2. 如果mp\_subprogramToHeadFile中包含以subprogramId为键值的项{
3. mp\_headFileShow[mp\_subprogramToHeadFile[subprogramId]]=**true**;
4. }
5. }

* 备注

在获取过程调用或函数调用时调用该函数，该函数可以将被调用的库程序所在的头文件进行标记，以便后续输出完整的头文件

### 获取常量列表

* 函数接口

1. **void** inputConstList(vector<\_Constant \*> &constList, vector<string> &constIdList, vector<string> &constTypeList, vector<string> &constValueList, \_SymbolTable\* symbolTable);

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| vector<\_Constant \*> &constList | AST的常量节点列表，可能是全局常量，也可能是某个子程序的常量 |
| vector<string> &constIdList | 存储常量标识符 |
| vector<string> &constTypeList | 存储常量类型 |
| vector<string> &constValueList | 存储常量值 |
| \_SymbolTable\* symbolTable | 符号表指针 |

* 函数伪代码

1. **void** inputConstList(vector<\_Constant \*> &constList, vector<string> &constIdList, vector<string> &constTypeList, vector<string> &constValueList, \_SymbolTable\* symbolTable){
2. 清空constIdList,constTypeList,constValueList;
3. 遍历AST的\_Constant节点{
4. 将常量标识符添加到constIdList中;
5. 将从符号表获取的常量类型添加到constTypeList中;
6. 从符号表获取常量值，要求是string形式，如果是负数，需要在前面添加负号;
7. 将常量值添加到constValueList中;
8. }
9. }

### 获取变量列表

* 函数接口

1. **void** inputVariantList(vector<\_Variant \*> variantList, vector<string> &variantIdList, vector<string> &variantTypeList, vector< vector<**int**> > &arraySizeList, \_SymbolTable\* symbolTable)

* 返回值类型

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| vector<\_Variant \*> variantList | AST的变量节点列表，可能是全局变量，也可能是某个子程序的变量 |
| vector<string> &variantIdList | 存储变量标识符 |
| vector<string> &variantTypeList | 存储变量类型 |
| vector< vector<**int**> > &arraySizeList | 存储数组各维大小 |
| \_SymbolTable\* symbolTable | 符号表指针 |

* 函数伪代码

1. **void** inputVariantList(vector<\_Variant \*> variantList, vector<string> &variantIdList, vector<string> &variantTypeList, vector< vector<**int**> > &arraySizeList, \_SymbolTable\* symbolTable){
2. 清空variantList,variantIdList,variantTypeList;
3. 遍历AST的\_Variant节点{
4. 将变量标识符添加到variantIdList中;
5. 将从符号表获取的变量类型添加到variantTypeList中;
6. if(是数组)
7. 将从符号表获取的数组各维的大小添加到arraySizeList中;
8. }
9. }

### 获取子程序接口声明列表

* 函数接口

1. **void** inputSubproDecList()

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** inputSubproDecList(){
2. 查主符号表，遍历所有子程序名{
3. subproDec.returnType=从符号表获取的返回值类型
4. id=子程序名
5. 从符号表获取每一个参数{
6. subproDec.v\_paraIdList.push\_back(参数标识符)
7. 从符号表获取参数是引用参数还是传值参数;
8. **if**(参数为传引用)
9. subproDec.v\_isParaRef.push\_back(**true**);
10. **else**
11. subproDec.v\_isParaRef.push\_back(**false**);
12. subproDec.v\_paraTypeList.push\_back(从符号表获得的参数类型);
13. }
14. 将subproDec加入到子程序声明列表中;
15. }
16. }

### 获取函数调用

* 函数接口

1. **void** inputFunctionCall(\_FunctionCall \*functionCallNode, string &functionCall, int mode=0)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_FunctionCall \*functionCallNode | AST的\_FunctionCall节点，即函数调用 |
| string &functionCall | 用于存放函数调用获取结果 |
| int mode | mode=0表示按照C语言输出函数调用，mode=1表示按照PASCAL-S语言输出函数调用，后者主要是提供给语义分析报错的接口 |

* 函数伪代码

1. **void** inputFunctionCall(\_FunctionCall \*functionCallNode, string &functionCall, int mode){
2. 调用inputHeadFileList标记头文件;
3. functionCall=AST的\_FunctionCall节点的函数名信息;
4. if(该函数带参数)
5. functionCall+="(";
6. 遍历AST的\_FunctionCall节点的实参列表节点{
7. **if**(不是第一个实参)
8. functionCall+=", ";
9. 查符号表获取当前实参对应的形参是否是引用参数，保存在布尔变量isRefered中
10. 调用inputExpression(mode, isRefered)获取当前的实参表达式;
11. functionCall+=当前的实参表达式;
12. }
13. if(该函数带参数)
14. functionCall+=")";
15. }

* 备注

伪代码中调用了inputExpression函数，该函数用于获取表达式

### 获取表达式

* 函数接口

1. **int** inputExpression(\_Expression \*expressionNode, string &expression, **int** mode=0, **bool** isReferedActualPara=**false**);

* 返回值

int 表示当前获取的表达式的一级运算符的优先级

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Expression \*expressionNode | AST的\_Expression节点 |
| string &expression | 存储表达式 |
| int mode | mode=0表示按照C语言输出表达式，mode=1表示按照PASCAL-S语言输出表达式，后者主要是提供给语义分析报错的接口 |
| bool isReferedActualPara | true表示当前表达式作为程序调用的实参，且对应的形参为引用参数，false则表示不是上述的情况 |

* 函数伪代码

1. //表达式
2. //需要根据操作符之间的优先级关系添加括号，所以定义了一个flag，用来指示子表达式用到的操作符类型
3. //变量引用、整数、浮点数、函数调用、复合表达式
4. **int** inputExpression(\_Expression \*expressionNode, string &expression, int mode, bool isReferedActualPara){
5. //返回值用于表示是否需要加括号
6. **if**(是变量标识符){
7. string variantRef;
8. 调用inputVariantRef获取变量引用(mode, isReferedActualPara)，结果保存在variantRef中
9. 将variantRef拼接到原有expression后面
10. **return** 0;//无运算符，所以优先级为0
11. }
12. **else** **if**(是整数){
13. 将整数拼接到原有expression后面;
14. **return** 0;
15. }
16. **else** **if**(是浮点数){
17. 将浮点数拼接到原有expression后面;
18. **return** 0;
19. }
20. **else** **if**(是函数调用){
21. string functionCall;
22. 调用inputHeadFileList标记头文件;
23. 调用inputFunctionCall获取函数调用，结果保存在functionCall中
24. 将functionCall拼接到原有expression后面;
25. **return** 0;
26. }
27. **else** **if**(是复合表达式){
28. **if**(是单目运算符){
29. 调用inputExpression(mode)获得子表达式及其优先级
30. **if**(是添加括号){
31. 在子表达式两边添加括号之后，拼接到原有expression后面;
32. **return** 0;//单目运算符优先级最高，为4
33. }
34. **if**(是取相反数)
35. 将"-"拼接到原有expression后面;
36. **else** **if**(是取非)
37. 将"!"拼接到原有expression后面;
38. **if**(mode==0 && 子表达式优先级大于0)
39. 在子表达式两边添加括号之后，拼接到原有expression后面;
40. **else**
41. 将子表达式拼接到原有expression后面;
42. **return** 4;//单目运算符优先级最高，为4
43. }
44. **else** **if**(是"mulop"){//"mulop"包括\*、/、mod、div、and
45. 调用inputExpression(mode)获取左子表达式及其优先级
46. **if**(左子表达式优先级为1或2)
47. 在左子表达式两边添加括号之后，拼接到原有expression后面;
48. **else**
49. 将左子表达式拼接到原有exprssion后面;
50. 将运算符拼接到原有expression后面;
51. 调用inputExpression(mode)获取右子表达式及其优先级;
52. **if**(mode == 0 && 右子表达式优先级为1或2)
53. 在右子表达式两边添加括号之后，拼接到原有expression后面;
54. **else**
55. 将右子表达式拼接到原有exprssion后面;
56. **return** 3;//"mulop"优先级次于单目运算符,为3
57. }
58. **else** **if**(是"addop"){//"addop"包括+、-、or
59. 调用inputExpression(mode)获取左子表达式及其优先级;
60. **if**(左子表达式优先级为1)
61. 在左子表达式两边添加括号之后，拼接到原有expression后面;
62. **else**
63. 将左子表达式拼接到原有exprssion后面;
64. 将运算符拼接到原有expression后面;
65. 调用inputExpression(mode)获取右子表达式及其优先级;
66. **if**(mode == 0 && 右子表达式优先级为1)
67. 在右子表达式两边添加括号之后，拼接到原有expression后面;
68. **else**
69. 将右子表达式拼接到原有exprssion后面;
70. **return** 2;//"addop"优先级次于"mulop"，为2
71. }
72. **else** **if**(是"relop"){//"relop"包括=、<>、<、<=、>、>=
73. string tmp;
74. 调用inputExpression(mode)获取左子表达式
75. 将左子表达式拼接到原有expression后面;
76. 将运算符拼接到原有expression后面;
77. tmp="";
78. 调用inputExpression(mode)获取右子表达式
79. 将右子表达式拼接到原有exprssion后面;
80. **return** 1;//"relop"优先级次于"addop"，为1
81. }
82. }
83. }

* 备注

需要根据当前运算符和子表达式运算符的优先级关系，来决定是否需要在子表达式两边添加括号，所以设置了一个flag，用于表示表达式中所包含运算符的优先级。

变量引用、整数、浮点数、函数调用没有运算符，优先级为0；单目运算符优先级为4，mulop优先级为3，addop优先级为2，relop优先级为1。

### 获取变量引用

* 函数接口

1. **void** inputVariantRef(\_VariantReference \*variantRefNode, string &variantRef, **int** mode=0, **bool** isReferedActualPara=**false**);

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_VariantReference \*variantRefNode | AST的\_VariantReference节点 |
| string &variantRef | 保存变量引用 |
| int mode | mode=0表示按照C语言输出变量引用，mode=1表示按照PASCAL-S语言输出变量引用，后者主要是提供给语义分析报错的接口 |
| bool isReferedActualPara | true表示当前变量引用作为程序调用的实参，且对应的形参为引用参数，false则表示不是上述的情况 |

* 函数伪代码

1. **void** inputVariantRef(\_VariantReference \*variantRefNode, string &variantRef,**int** mode, **bool** isReferedActualPara){
2. variantRef=\_VariantReference节点的变量标识符信息;
3. if (mode == 0 && 变量引用的种类是变量或者数组元素) {
4. 查符号表，检查当前标识符是否是引用参数;
5. bool isRefered = checkIsReferedPara(codeGenerateCurrentSymbolTable, variantRefNode->variantId.first);
6. if (是引用参数) {
7. if (!isReferedActualPara)
8. variantRef前添加"\*";
9. }
10. else {//不是引用参数
11. if (isReferedActualPara)
12. variantRef前添加"&";
13. }
14. }
15. //如果是函数，一定是右值，如果是左值，代码逻辑保证了不会调用该函数
16. //通过了语义分析，保证了该函数形参个数一定为0
17. **if** (需要输出目标代码 && \_VariantReference的种类是函数调用)
18. 将"()"拼接到variantRef后面;
19. **if** (\_VariantReference的种类不是数组)
20. **return**;
21. //这里得查符号表，找出数组定义时每一维的下界
22. **if**(mode==0){//生成C代码
23. 查符号表，找到数组定义时每一维的下界{
24. 调用inputExpression(mode)获取当前维的下标表达式;
25. string delta;//需要添加在下标表达式后面的偏移量部分
26. **if** (数组当前维下界>0)
27. delta = " - " + 数组当前维下界;
28. **else** **if** (数组当前维下界 < 0)
29. delta = " + " + 数组当前维下界的绝对值;
30. **else**//如果数组当前维下界刚好等于0
31. delta = "";
32. 将delta拼接到原有下标表达式后面;
33. 在表达式两端添加中括号后，拼接到原有variantRef后面;
34. }
35. }
36. **else**{//生成PASCAL-S代码
37. 将"["拼接到原有variantRef后面;
38. 查符号表，找到数组定义时每一维的下界{
39. 调用inputExpression(mode)获取当前维的下标表达式;
40. **if**(不是第一维)
41. 将", "拼接到原有variantRef后面;
42. 将下标表达式拼接到原有variantRef后面;
43. }
44. 将"]"拼接到原有variantRef后面;
45. }
46. }

### 获取语句列表

* 函数接口

1. **void** inputStatementList(\_Statement \*statementNode, vector< pair<string,**int**> > &v\_statementList, **int** retract, **int** flag=0)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Statement \*statementNode | AST的\_Statement节点 |
| vector< pair<string,**int**> > &v\_statementList | 保存获取的语句列表结果 |
| **int** retract | 当前新增语句应有的缩进（制表符的个数） |
| **int** flag | 是否需要在compound语句两侧添加花括号，如果flag==0表示需要，否则表示不需要 |

* 函数伪代码

1. **void** inputStatementList(\_Statement \*statementNode, vector< pair<string,**int**> > &v\_statementList, **int** retract, **int** flag=0){
2. **if**(是"compound"语句){
3. **if**(flag==0)
4. 添加"{"到语句列表中，缩进值为retract-1;
5. 遍历\_Compound节点的子语句列表
6. 调用inputStatementList获取复合语句列表（retract+1）;
7. **if**(flag==0)
8. 添加"}"到语句列表中，缩进值为retract-1;
9. }
10. **else** **if**(是"repeat"语句){
11. 添加do关键字到语句列表中，缩进值为retract;
12. 调用inputStatementList获取循环体语句（retract+1）;
13. 调用inputExpression获取条件表达式;
14. 将条件表达式取非后与while关键字组合成一条语句，添加到语句列表中，缩进值为retract;
15. }
16. **else** **if**(是"while"语句){
17. 调用inputExpression获取条件表达式;
18. 将条件表达式与while关键字组合成一条语句，添加到语句列表中，缩进值为retract;
19. 调用inputStatementList获取循环体语句（retract+1）;
20. }
21. **else** **if**(是"for"语句){
22. 调用inputExpression获取初值表达式;
23. 调用inputExpression获取终值表达式;
24. 将初值表达式、终止表达式、for关键字等组合成一条语句，添加到语句列表中，缩进值为retract;
25. 调用inputStatementList获取循环体语句（retract+1）;
26. }
27. **else** **if**(是"if"语句){
28. 调用inputExpression获取条件表达式;
29. 将条件表达式和if关键字组合成一条语句，添加到语句列表中，缩进值为retract;
30. 调用inputStatementList获取then语句体（retract+1）;
31. **if**(包含**else**部分){
32. 添加else关键字到语句列表中，缩进值为retract；
33. 调用inputStatementList获取else语句体（retract+1）;
34. }
35. }
36. **else** **if**(是"assign"语句){
37. 调用inputVariantRef获取左值变量引用;
38. 调用inputExpression获取右值表达式;
39. 将左值变量引用和右值表达式拼接成赋值语句，添加到语句列表中，缩进值为retract;
40. }
41. **else** **if**(是"procedure"语句，即过程调用){
42. 调用inputHeadFileList标记头文件;
43. 调用checkAndInputLibrarySubprogram检查并获取库函数;
44. if(是库函数)
45. return;
46. 遍历所有的实参表达式{
47. 查符号表获取当前实参对应的形参是否是引用参数，保存在布尔变量isRefered中
48. 调用inputExpression获取实参表达式(mode=0, isRefered);
49. }
50. 将过程名和所有实参表达式拼接成一条过程调用语句，并添加到语句列表中，缩进值为retract;
51. }
52. }

### 获取子程序定义

* 函数接口

1. **void** inputSubproDef(\_FunctionDefinition\* functionDefinitionNode)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_FunctionDefinition\* functionDefinitionNode | AST的\_FunctionDefinition节点 |

* 函数伪代码

1. **void** inputSubproDef(\_FunctionDefinition\* functionDefinitionNode){
2. **struct** subproDef tmp;
3. 调用inputConstList获取常量定义列表，添加到tmp中;
4. 调用inputVariantList获取变量定义列表，添加到tmp中;
5. 调用inputStatementList获取程序体语句列表，添加到tmp中;
6. 将tmp添加到子程序定义列表中;
7. }

### 获取子程序体列表

* 函数接口

1. **void** inputSubproDefList(\_SubProgram\* subProgramNode)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_SubProgram\* subProgramNode | AST的\_SubProgram节点 |

* 函数伪代码

1. **void** inputSubproDefList(\_SubProgram\* subProgramNode){
2. 遍历子程序定义列表{
3. 利用当前子程序名，从符号表获取当前子程序对应的字符号表指针;
4. codeGenerateCurrentSymbolTable定位到当前子程序的子符号表;
5. 调用inputSubproDef获取当前子程序的定义;
6. }
7. }

### 获取原PASCAL主程序头对应到的C程序头及程序体

* 函数接口

1. **void** inputSubMainFunction(\_Program\* ASTRoot);//获取原PASCAL主程序头对应到的C程序头，以及主程序体

* 返回值

无

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Program\* ASTRoot | 抽象语法树根节点 |

* 函数伪代码

1. **void** inputSubMainFunction(\_Program\* ASTRoot) {
2. subMainFunctionDeclaration = "void " + ASTRoot->programId.first + "()";//用void关键字和主程序名拼接成一个主函数头
3. 调用inputStatementList获取主函数语句体(retract=1,flag=1即不添加大括号);
4. }

### 根据类型获取输入输出格式控制符

* 函数接口

1. string getOutputFormat(string type);

* 返回值

string，获得的格式控制符

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string type | 基本类型 |

* 函数伪代码

1. string getOutputFormat(string type) {
2. **if** (type == "integer")
3. **return** "%d";
4. **if** (type == "real")
5. **return** "%f";
6. **if** (type == "char")
7. **return** "%c";
8. **if** (type == "boolean")
9. **return** "bool";
10. cout << "[getOutputFormat] type error" << endl;
11. **return** "";
12. }

### 检查并获取库程序

* 函数接口

1. **bool** checkAndInputLibrarySubprogram(\_ProcedureCall\* procedureCall, vector< pair<string, **int**> > &v\_statementList, **int** retract);

* 返回值

bool，如果检查到是库程序，则返回true，表示该函数内部已完成库程序调用的获取，如果检查到不是库程序，则返回false，表示该函数返回后，还需进一步获取程序调用的相关信息

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_ProcedureCall\* procedureCall | 过程调用节点 |
| vector< pair<string, int> > &v\_statementList | 语句列表 |
| int retract | 缩进（制表符个数） |

* 函数伪代码

1. **bool** checkAndInputLibrarySubprogram(\_ProcedureCall\* procedureCall, vector< pair<string, **int**> > &v\_statementList, **int** retract) {
2. //返回值表示是否是库程序调用
3. **if** (是exit过程调用) {
4. 直接将"return;"加入语句列表，缩进值为retract;
5. }
6. **else** **if** (是write或writeln过程调用) {
7. **if** (是writeln过程调用且实参个数为0) {
8. 直接将"printf(\"\\n\");"加入语句列表，缩进值为retract;
9. **return** **true**;
10. }
11. **for** (**int** i = 0; i < procedureCall->actualParaList.size(); i++) {
12. string expression, typeFormat;
13. 调用inputExpression获取实参表达式;
14. 调用getOutputFormat获取该实参表达式的类型对应的格式控制符;
15. **if** (实参表达式的类型是**bool**类型) {
16. 将之前已有的实参表达式及其格式控制符拼成一条printf语句;
17. 将printf语句加入语句列表，缩进值为retract;
18. 将"{"加入语句列表，缩进值为retract;
19. v\_statementList.push\_back(make\_pair("if(" + expression + ")", retract + 1));
20. v\_statementList.push\_back(make\_pair("printf(\"true\");", retract + 2));
21. v\_statementList.push\_back(make\_pair("else", retract + 1));
22. v\_statementList.push\_back(make\_pair("printf(\"false\");", retract + 2));
23. 将"}"加入语句列表，缩进值为retract;
24. if(当前是最后一个实参 && 是writeln过程)
25. 将输出换行的语句加入到语句列表中，缩进值为retract;
26. }
27. }
28. 将剩余的实参表达式及其格式控制符拼成一条printf语句，如果是“writeln”，还需在最后加入换行符;
29. 将printf语句加入语句列表，缩进值为retract;
30. }
31. **else** **if** (是read过程调用) {
32. **for** (**int** i = 0; i < procedureCall->actualParaList.size(); i++) {
33. 调用inputExpression获取实参表达式(mode=0, isReferedActualPara=true);
34. 调用getOutputFormat获取该实参表达式的类型对应的格式控制符;
35. **if**(实参表达式的类型是**bool**类型)
36. 格式控制符设为"%d";
37. }
38. 将所有实参表达式、格式控制符拼成一条scanf语句;
39. 将scanf语句加入语句列表，缩进值为retract;
40. }
41. **else**
42. **return** **false**;
43. **return** **true**;
44. }

## 输出代码的函数设计

### 输出头文件

* 函数接口

1. **void** outputHeadFileList()

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** outputHeadFileList(){
2. **for**(遍历mp\_headFileShow){//遍历
3. **if**(属性值为**true**){//如果涉及该头文件的过程或函数出现
4. cout << "#include<" << it->key << ">" << endl;//输出头文件
5. 属性值设为**false**;
6. }
7. }
8. 无条件输出‘#include<stdbool.h>’头文件
9. }

### 输出常数列表

* 函数接口

1. **void** outputConstList(vector<string> &constIdList, vector<string> &constTypeList, vector<string> &constValueList, **int** retract=0)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| vector<string> &constIdList | 常数标识符列表 |
| vector<string> &constTypeList | 常数类型列表 |
| vector<string> &constValueList | 常数取值列表 |
| **int** retract | 语句前的制表符个数（缩进控制） |

* 函数伪代码

1. **void** outputConstList(vector<string> &constIdList, vector<string> &constTypeList, vector<string> &constValueList, **int** retract=0){
2. **for**(遍历这三个列表){
3. 输出retract个制表符;
4. cout << "const " << constTypeList[i] << " " <<  constIdList[i] << "=" << constValueList[i] << ";" <<endl;//将三个列表的信息组装成常量定义并输出
5. }
6. }

### 输出变量列表

* 函数接口

1. **void** outputVariantList(vector<string> &variantIdList, vector<string> &variantTypeList, vector< vector<**int**> > &arraySizeList, **int** retract=0)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| vector<string> &variantIdList | 变量标识符列表 |
| vector<string> &variantTypeList | 变量类型列表 |
| vector< vector<**int**> > &arraySizeList | 变量各维大小列表 |
| **int** retract | 语句前的制表符个数（缩进控制） |

* 函数伪代码

1. **void** outputVariantList(vector<string> &variantIdList, vector<string> &variantTypeList, vector< vector<**int**> > &arraySizeList, **int** retract=0){
2. 遍历这三个列表{
3. 输出retract个制表符;
4. cout << variantTypeList[i] << " " << variantIdList[i];
5. 遍历数组各维大小
6. cout << "[" << arraySizeList[i][j] << "]";
7. cout << ";" << endl;  //将三个列表的信息组装成变量定义并输出
8. }
9. }

### 输出子程序接口声明

* 函数接口

1. **void** outputSubproDec(subproDec &tmp,**int** flag=0)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| subproDec &tmp | 子程序声明结构体 |
| **int** flag | flag==0时表示接口声明,flag==1时表示定义时的子程序头,前者比后者在尾部多一个分号 |

* 函数伪代码

1. **void** outputSubproDec(subproDec &tmp, **int** flag) {//flag==0时表示接口声明,flag==1时表示定义时的子程序头
2. 输出返回值类型、空格、子程序名、左括号;
3. 遍历形参列表 {
4. **if** (不是第一个形参)
5. 输出逗号、空格
6. 输出形参类型、空格;
7. **if** (当前形参是传引用参数)
8. 输出"\*";
9. 输出形参标识符;
10. }
11. 输出右括号;
12. **if** (flag == 0)
13. 输出分号;
14. 输出换行;
15. }

### 输出子程序接口声明列表

* 函数接口

1. **void** outputSubproDecList()

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** outputSubproDecList(){
2. 输出原PASCAL主程序对应C的程序头;
3. vector< **struct** subproDec > &vec = v\_subproDecList;
4. 遍历子程序定义列表vec
5. 调用outputSubproDec输出每一个子程序声明（最后添加分号）;
6. }

### 输出语句

* 函数接口

1. **void** outputStatement(pair<string,**int**> &tmp)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| pair<string,**int**> &tmp | 语句及其前的制表符个数 |

* 函数伪代码

1. **void** outputStatement(pair<string,**int**> &tmp){
2. 输出tmp.second个制表符;
3. 输出语句;
4. 输出换行;
5. }

### 输出语句列表

* 函数接口

1. **void** outputStatementList(vector< pair<string,**int**> > &vec)

* 返回值

void

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| vector< pair<string,**int**> > &vec | 语句列表 |

* 函数伪代码

1. **void** outputStatementList(vector< pair<string,**int**> > &vec){
2. 遍历语句列表中的每一条语句
3. 调用outputStatement输出每一条语句;
4. vec.clear();//清空
5. }

### 输出子程序定义列表

* 函数接口

1. **void** outputSubproDefList()

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** outputSubproDefList(){
2. 输出pascal主程序对应的C程序头、换行;
3. 输出左大括号、换行;
4. 输出pascal主程序对应的C程序体;
5. 输出右大括号、换行;
6. 遍历子程序定义列表{
7. 调用outputSubproDec输出子程序头（后面不需要添加分号）;
8. 输出左大括号、换行;
9. 调用outputConstList输出全局常量定义列表;
10. 调用outputVariantList输出全局变量定义列表;
11. 调用outputStatementList输出程序体语句列表;
12. 输出右大括号、换行、换行;
13. }
14. }

### 输出main函数体

* 函数接口

1. **void** ouputMain()

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** ouputMain(){
2. cout << "int main()\n{\tmain\_function();\n\treturn 0;\n}\n";
3. }

## 其它函数设计

### 代码生成之前的初始化

* 函数接口

1. **void** beforeCodeGenerate()

* 返回值

void

* 参数列表

无

* 函数伪代码

1. **void** beforeCodeGenerate(){
2. 初始化过程、函数到C程序头文件的映射;
3. 初始化头文件是否出现的映射;
4. }

### 判断数字是否相等（变参）

* 函数接口

1. **bool** isEqual(**int** num,...);

* 返回值

bool，当所有数字相等时返回true，否则返回false

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| **int** num | 后续参数个数 |
| ... | 不定量整数参数 |

* 函数伪代码

1. **bool** isEqual(**int** num,...){//变参列表,num表示参数个数
2. **va\_list** argp;
3. **int** para;
4. va\_start(argp, num);
5. **int** pre=va\_arg(argp, **int**);
6. **for**(**int** argno=1; argno<num; argno++){
7. para = va\_arg(argp, **int**);
8. **if**(para!=pre)
9. **return** **false**;//存在不相等的数
10. }
11. va\_end(argp);
12. **return** **true**;//所有数相等
13. }

### 将PASCAL-S的类型关键字转化为C的类型关键字

* 函数接口

1. string transformType(string pascalType);

* 返回值

string，转化后的C类型关键字

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string pascalType | 转化前的PASCAL-S的类型关键字 |

* 函数伪代码

1. string transformType(string pascalType) {
2. **if** (pascalType == "integer")
3. **return** "int";
4. **if** (pascalType == "real")
5. **return** "float";
6. **if** (pascalType == "boolean")
7. **return** "bool";
8. **if** (pascalType == "char")
9. **return** "char";
10. **if** (pascalType == "")
11. **return** "";
12. cout << "[transformType] pascalType error" << endl;
13. **return** "";
14. }

### 将PASCAL-S的运算符转化为C的运算符

* 函数接口

1. string transformOpertion(string operation, **int** mode=0);

* 返回值

string，mode==0时表示转化后的C运算符，mode!=0时表示原样返回，即认为PASCAL-S的运算符

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string operation | 转化前的PASCAL-S运算符 |
| int mode | mode==0表示转化，mode!=0表示不转化，也就是原样返回 |

* 伪代码

1. string transformOpertion(string operation,**int** mode) {
2. **if** (mode != 0)
3. **return** operation;
4. **if** (operation == "not")
5. **return** "!";
6. **if** (operation == "=")
7. **return** "==";
8. **if** (operation == "<>")
9. **return** "!=";
10. **if** (operation == "or")
11. **return** "||";
12. **if** (operation == "div")
13. **return** "/";
14. **if** (operation == "mod")
15. **return** "%";
16. **if** (operation == "and")
17. **return** "&&";
18. **return** operation;
19. }

### 代码生成对外接口

* 函数接口

1. **void** codeGenerate(\_Program \*ASTRoot, string outName);

* 返回值

无

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| \_Program \*ASTRoot | 抽象语法树的根节点 |
| string outName | 保存输出C代码的文件名 |

* 伪代码

1. **void** codeGenerate(\_Program \*ASTRoot, string outName) {
2. 打开outName，并将其关联到输出文件流fout;
3. codeGenerateCurrentSymbolTable定位到主符号表;
4. 调用beforeCodeGenerate做代码生成之前的初始化工作;
5. 调用inputSubproDecList获取子程序声明列表;
6. 调用inputConstList获取主程序常量定义列表;
7. 调用inputVariantList获取主程序变量定义列表;
8. 调用inputSubproDefList获取子程序定义列表;
9. 调用inputSubMainFunction(ASTRoot)获取原PASCAL主程序对应C的程序头和语句体;
11. fout << "//Head files" << endl; //输出头文件区前缀注释
12. 调用outputHeadFileList输出头文件;
13. fout << endl;
15. fout << "//Overall constant definiton" << endl; //输出全局常量区前缀注释
16. 调用outputConstList输出全局常量定义;
17. fout << endl;
19. fout << "//Overall variable definition" << endl; //输出全局变量区前缀注释
20. 调用outputVariantList输出全局变量定义;
21. fout << endl;
23. fout << "//Subprogram declaration" << endl; //输出子程序声明区前缀注释
24. 调用outputSubproDecList输出子程序声明;
25. fout << endl;
27. fout << "//Main function" << endl; //输出主程序区前缀注释
28. 调用outputMain输出主函数;
29. fout << endl;
31. fout << "//Subprogram definition" << endl; //输出子程序定义区前缀注释
32. 调用outputSubproDefList输出子程序定义;
34. fout.close();
35. }

### 提供给语义分析的接口说明

以下三个接口提供给了语义分析的错误处理部分，以输出更加详细易读的报错信息：

1. **extern** **void** inputFunctionCall(\_FunctionCall \*functionCallNode, string &functionCall, **int** mode=0);//获取函数调用
2. **extern** **int** inputExpression(\_Expression \*expressionNode, string &expression, **int** mode=0, **bool** isReferedActualPara=**false**);//获取表达式
3. **extern** **void** inputVariantRef(\_VariantReference \*variantRefNode, string &variantRef, **int** mode=0, **bool** isReferedActualPara=**false**);//获取变量引用

其中都涉及到了mode参数，如果mode=0，表示是最终代码生成的情形，需要转化为C代码输出，如果mode!=0，表示是语义分析调用的情形，需要按照PASCAL-S的源代码输出。

语义分析程序在调用时，需要指定mode参数不为0。

在实现inputExpression时，PASCAL-S代码和C代码的主要区别在于运算符，所以又需要调用下列函数：

1. string transformOpertion(string operation, **int** mode=0);//将pascal的operation转化为c的operation

同时也要考虑到自加括号的情形，所以在添加括号是，也要判断mode是否为0。

在实现inputVariantRef时，除了数组下标引用列表的形式需要判断mode外，也要注意无参调用的情形，PASCAL-S无参调用时只需要函数名，而C还需要在函数名后面添加一对括号。

最后要注意这三个函数的实现中，在相互调用时要加上mode参数，以保证完整地按照PASCAL-S代码输出，或者完整地按照C代码输出。