# 目录

[1 目录 1](#_Toc513585272)

[2 语法分析详细设计 3](#_Toc513585273)

[2.1 语法分析方法概述 3](#_Toc513585274)

[2.2 工具介绍 3](#_Toc513585275)

[2.2.1 YACC概述 3](#_Toc513585276)

[2.2.2 YACC规则 4](#_Toc513585277)

[2.2.3 YACC冲突解决 4](#_Toc513585278)

[2.3 语法规则的定义 4](#_Toc513585279)

[2.4 语法分析树涉及的数据结构 7](#_Toc513585280)

[2.4.1 结点的数据类型 7](#_Toc513585281)

[2.4.2 错误处理相关全局变量 7](#_Toc513585282)

[2.4.3 错误处理相关函数声明 7](#_Toc513585283)

[2.5 构建语法树过程描述 8](#_Toc513585284)

[2.6 错误处理相关设计 9](#_Toc513585285)

[2.6.1 错误处理种类 9](#_Toc513585286)

[2.6.2 错误处理相关产生式处理 9](#_Toc513585287)

[2.6.3 错误处理相关函数设计 10](#_Toc513585288)

[3 普通语法分析树转AST详细设计 11](#_Toc513585289)

[3.1 AST节点类定义 11](#_Toc513585290)

[3.1.1 节点类定义代码 11](#_Toc513585291)

[3.1.2 节点类定义细节 16](#_Toc513585292)

[3.1.2.1 \_Expression的种类设计 16](#_Toc513585293)

[3.1.2.2 \_VariantRerence区分是否数组 17](#_Toc513585294)

[3.1.2.3 \_Statement类之间的关系 17](#_Toc513585295)

[3.1.2.4 \_Type区分是否数组 18](#_Toc513585296)

[3.1.2.5 \_FormalParameter区分是否是引用参数 18](#_Toc513585297)

[3.1.2.6 \_Constant常量定义细节 18](#_Toc513585298)

[3.1.2.7 不同节点类的类型区别 18](#_Toc513585299)

[3.2 函数接口设计 18](#_Toc513585300)

[3.2.1 辅助函数 18](#_Toc513585301)

[3.2.1.1 字符串转整型 18](#_Toc513585302)

[3.2.1.2 字符串转浮点型 19](#_Toc513585303)

[3.2.2 转化为AST的函数 19](#_Toc513585304)

[3.2.2.1 获取标识符列表 19](#_Toc513585305)

[3.2.2.2 获取数组上下界列表 19](#_Toc513585306)

[3.2.2.3 获取类型 20](#_Toc513585307)

[3.2.2.4 获取变量定义 20](#_Toc513585308)

[3.2.2.5 获取变量定义列表 20](#_Toc513585309)

[3.2.2.6 获取常量定义节点具体信息 20](#_Toc513585310)

[3.2.2.7 获取常量定义节点 21](#_Toc513585311)

[3.2.2.8 获取常量定义节点列表 21](#_Toc513585312)

[3.2.2.9 获取形式参数的具体信息 21](#_Toc513585313)

[3.2.2.10 区别形式参数 22](#_Toc513585314)

[3.2.2.11 获取形式参数 22](#_Toc513585315)

[3.2.2.12 获取形式参数节点列表 22](#_Toc513585316)

[3.2.2.13 获取子程序头 23](#_Toc513585317)

[3.2.2.14 获取factor 23](#_Toc513585318)

[3.2.2.15 获取term 23](#_Toc513585319)

[3.2.2.16 获取simple expression 24](#_Toc513585320)

[3.2.2.17 获取expression 24](#_Toc513585321)

[3.2.2.18 获取表达式列表 24](#_Toc513585322)

[3.2.2.19 获取变量引用节点 24](#_Toc513585323)

[3.2.2.20 获取变量引用节点列表 25](#_Toc513585324)

[3.2.2.21 获取else部分语句 25](#_Toc513585325)

[3.2.2.22 获取过程调用 25](#_Toc513585326)

[3.2.2.23 获取各种类型的语句 25](#_Toc513585327)

[3.2.2.24 获取语句列表 26](#_Toc513585328)

[3.2.2.25 获取复合语句块 26](#_Toc513585329)

[3.2.2.26 获取子程序体 26](#_Toc513585330)

[3.2.2.27 获取函数定义 27](#_Toc513585331)

[3.2.2.28 获取子程序定义列表 27](#_Toc513585332)

[3.2.2.29 获取主程序体 27](#_Toc513585333)

[3.2.2.30 获取主程序头 27](#_Toc513585334)

[3.2.2.31 获取整个程序 28](#_Toc513585335)

[3.3 各函数伪代码 28](#_Toc513585336)

[3.3.1 辅助函数 28](#_Toc513585337)

[3.3.1.1 字符串转整型 28](#_Toc513585338)

[3.3.1.2 字符串转浮点型 28](#_Toc513585339)

[3.3.2 转化为AST的函数 29](#_Toc513585340)

[3.3.2.1 获取标识符列表 29](#_Toc513585341)

[3.3.2.2 获取数组上下界列表 29](#_Toc513585342)

[3.3.2.3 获取类型 30](#_Toc513585343)

[3.3.2.4 获取变量定义 30](#_Toc513585344)

[3.3.2.5 获取变量定义列表 30](#_Toc513585345)

[3.3.2.6 获取常量定义节点具体信息 31](#_Toc513585346)

[3.3.2.7 获取常量定义节点 32](#_Toc513585347)

[3.3.2.8 获取常量定义节点列表 32](#_Toc513585348)

[3.3.2.9 获取形式参数的具体信息 32](#_Toc513585349)

[3.3.2.10 区别形式参数 33](#_Toc513585350)

[3.3.2.11 获取形式参数 33](#_Toc513585351)

[3.3.2.12 获取形式参数节点列表 33](#_Toc513585352)

[3.3.2.13 获取子程序头 34](#_Toc513585353)

[3.3.2.14 获取factor 34](#_Toc513585354)

[3.3.2.15 获取term 36](#_Toc513585355)

[3.3.2.16 获取simple expression 36](#_Toc513585356)

[3.3.2.17 获取expression 37](#_Toc513585357)

[3.3.2.18 获取表达式列表 37](#_Toc513585358)

[3.3.2.19 获取变量引用节点 37](#_Toc513585359)

[3.3.2.20 获取变量引用节点列表 38](#_Toc513585360)

[3.3.2.21 获取else部分语句 38](#_Toc513585361)

[3.3.2.22 获取过程调用 38](#_Toc513585362)

[3.3.2.23 获取各种类型的语句 39](#_Toc513585363)

[3.3.2.24 获取语句列表 40](#_Toc513585364)

[3.3.2.25 获取复合语句块 41](#_Toc513585365)

[3.3.2.26 获取子程序体 41](#_Toc513585366)

[3.3.2.27 获取函数定义 41](#_Toc513585367)

[3.3.2.28 获取子程序定义列表 41](#_Toc513585368)

[3.3.2.29 获取主程序体 42](#_Toc513585369)

[3.3.2.30 获取主程序头 42](#_Toc513585370)

[3.3.2.31 获取整个程序 42](#_Toc513585371)

[3.4 实现细节 43](#_Toc513585372)

[3.4.1 各节点类的行号设计 43](#_Toc513585373)

[3.4.2 左递归与列表反转 43](#_Toc513585374)

[3.4.3 常量值保存 44](#_Toc513585375)

[3.5 对外接口 44](#_Toc513585376)

[3.5.1 入口 44](#_Toc513585377)

[3.5.2 出口 44](#_Toc513585378)

# 语法分析详细设计

该部分利用YACC实现了生成普通语法树，并进行了错误报告、处理和恢复。

## 语法分析方法概述

第一步：运用Flex，将词法分析程序扫描源程序字符串、识别并生成的记号序列作为输入，语法分析阶段根据我们定制好的Pascal各个语法结构的产生式，利用Bison将其语法规则转化成自底向上的分析器，进而验证这个记号序列的合法性。若合法，则生成并输出完整的语法分析树，并转化为抽象语法树（AST）；若不合法，则对错误进行适当的恢复，向用户输出错误的性质以及发生位置。

第二步：语法树转化为AST的过程，具体见普通语法树转化为AST的详细设计。

## 工具介绍

### YACC概述

YACC（Yet Another Compiler Compiler），是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器）。YACC生成的编译器主要是用 C 语言编写的语法解析器（Parser），本来只在 Unix 系统上才有，但现在已普遍移植往 Windows 及其他平台。YACC源程序文件名约定为yacc.y，通过Bison工具的编译生成yacc.tab.cpp和yacc.tab.h两个文件。

### YACC规则

YACC的输入是巴科斯范式（BNF）表达的 LALR(1) 文法，输出是基于分析表驱动的编译器。

其源程序规则如下，与 Flex 非常相似：

1. %{
2. Declarations
3. %}
4. Definitions
5. %%
6. Translation rules
7. %%
8. User subroutines

Declarations 是 C 代码的声明与实现，YACC会将这部分代码直接拷贝到生成的 C 程序中，Definitions 是 YACC的辅助说明部分，目的是为翻译规则（Translation rules）服务。在该部分一般可以定义，Translation rules 部分为翻译规则，User subroutines 是用户自定义程序段。

### YACC冲突解决

移进/归约冲突时，执行移进动作，即移进先于归约；

归约/归约冲突时，用YACC源程序中第一个出现的产生式进行归约。

## 语法规则的定义

根据需求分析，我们对讲义上给出的文法产生式做出了一定的改进，并注释了改进的原因。以下产生式中，小写的字符串代表非终结符即非叶结点，反之大写的字符串以及字符常量是终结符相应的叶节点。

1 programstruct –> program\_head ';' program\_body '.' （1.1）

2 program\_head –> PROGRAM IDENTIFIER '(' idlist ')' （2.1）

3 program\_body –> const\_declarations var\_declarations subprogram\_declarations compound\_statement （3.1）

4 idlist –> idlist ',' IDENTIFIER （4.1）

| IDENTIFIER （4.2）

5 const\_declarations –> CONST const\_declaration ';' （5.1）

| ε （5.2）

6 const\_declaration –> const\_declaration ';' IDENTIFIER '=' const\_value （6.1）

| IDENTIFIER '=' const\_value （6.2）

//词法分析器对于常数的输出分为无符号常数和无符号常浮点数，且词法分析器也可以识别字符常量，即'字符'。故我们**对常量表达式const\_value的表达式作出改进**如下所示（其中UINUM表示无符号常整数，UFNUM表示无符号常浮点数，CHAR表示字符常量）：

7 const\_value –> '+' IDENTIFIER （7.1）

| '-' IDENTIFIER (7.2)

| IDENTIFIER (7.3)

| '+' UINUM (7.4)

| '-' UINUM (7.5)

| UINUM (7.6)

| '+' UFNUM (7.7)

| '-' UFNUM (7.8)

| UFNUM (7.9)

| CHAR (7.10)

8 var\_declarations –> VAR var\_declaration ';' (8.1)

| ε (8.2)

9 var\_declaration –> var\_declaration ';' idlist ':' type (9.1)

| idlist ':' type (9.2)

10 type –> TYPE (10.1)

| ARRAY '[' period ']' OF TYPE (10.2)

//词法分析器能够识别TYPE并将其分为integer, real, boolean和char，故**我们作出改进，将“简单类型”TYPE视为了终结符。**

11 period –> period ',' UINUM RANGEDOT UINUM (11.1)

| UINUM RANGEDOT UINUM (11.2)

12 subprogram\_declarations –> subprogram\_declarations subprogram ';' (12.1)

| ε (12.2)

13 subprogram –> subprogram\_head ';' subprogram\_body (13.1)

14 subprogram\_head –> PROCEDURE IDENTIFIER formal\_parameter (14.1)

| FUNCTION IDENTIFIER formal\_parameter ':' TYPE (14.2)

15 formal\_parameter –> '(' parameter\_list ')' (15.1)

| ε (15.2)

16 parameter\_list –> parameter\_list ';' parameter (16.1)

| parameter (16.2)

17 parameter –> var\_parameter (17.1)

| value\_parameter (17.2)

18 var\_parameter –> VAR value\_parameter (18.1)

19 value\_parameter –> idlist ':' TYPE (19.1)

20 subprogram\_body –> const\_declarations var\_declarations compound\_statement (20.1)

21 compound\_statement –> \_BEGIN statement\_list END (21.1)

22 statement\_list –> statement\_list ';' statement (22.1)

| statement (22.2)

//对于statement文法产生式，我们在实现基本要求的基础上，**增添了分别对应WHILE DO循环语句以及REPEAT UNTIL循环语句的文法产生式（23.5、 23.6）**。

23 statement –> variable ASSIGNOP expression (23.1)

| procedure\_call (23.2)

| compound\_statement (23.3)

| IF expression THEN statement else\_part (23.3)

| FOR IDENTIFIER ASSIGNOP expression TO expression DO statement (23.4)

| WHILE expression DO statement (23.5)

| REPEAT statement UNTIL expression (23.6)

| ε (23.7)

//不难知道，当处理if-then和if-then-else语句时，若不采用“最近最后匹配原则”，语法分析器将会发生臭名昭著的移进/归约冲突。Bison在处理这样的冲突时缺省的动作是进行移进，为了让冲突的解决过程更加清晰，我们定义了各个结点的优先级（注：最先定义的优先级最低，%left和%right分别代表左结合和右结合，而%nonassoc代表该记号在同一行内只能出现一次，否则发生语法错误）：

1. %left '+' '-' ADD
2. %left '\*' '/' MUL
3. %right UMINUS
4. %nonassoc LOWER\_THAN\_ELSE
5. %nonassoc ELSE
6. %nonassoc ONE
7. %nonassoc TWO
8. %nonassoc THREE

24 else\_part –> ELSE statement (24.1)

| %prec LOWER\_THAN\_ELSE (24.2)

25 variable –> IDENTIFIER id\_varpart (25.1)

26 id\_varpart –> '[' expression\_list ']' (26.1)

| ε (26.2)

27 procedure\_call –> IDENTIFIER (27.1)

| IDENTIFIER '(' expression\_list ')' (27.2)

28 expression\_list –> expression\_list ',' expression (28.1)

| expression (28.2)

29 expression –> simple\_expression RELOP simple\_expression (29.1)

| simple\_expression '=' simple\_expression (29.2)

| simple\_expression (29.3)

30 simple\_expression –> simple\_expression ADDOP term (30.1)

| simple\_expression '-' term (30.2)

| term (30.3)

31 term –> term MULOP factor (31.1)

| factor (31.2)

//**由于字符常量也是可以参加运算的，因此作出改进，增添文法产生式32.8。**

32 factor –> UINUM (32.1)

| UFNUM (32.2)

| variable (32.3)

| IDENTIFIER '(' expression\_list ')' (32.4)

| '(' expression ')' (32.5)

| NOT factor (32.6)

| '-' factor (32.7)

| CHAR (32.8)

## 语法分析树涉及的数据结构

### 结点的数据类型

首先，我们重新定义了Flex和Bison共享的yylval的数据类。在词法分析阶段将记号以该类的形式传递给语法分析，从而在语法分析树的构建过程中，各个结点具有统一封装的特性。

1. //重新定义属性类型
2. **class** Type{
3. **public**:
4. string str;//终结符号的具体属性
5. string token;//终结符号或非终结符号本身的名称
6. **int** lineNumber;//终结符号的行号，参照语法分析指导.docx
7. vector<Type\*> children; //对应产生式下面的结点
9. Type(){}
10. Type(string typ, string name, **int** ln): str(typ), token(name), lineNumber(ln){}
11. Type(string name, vector<Type\*> cdn): token(name), children(cdn){}
12. };
13. #define YYSTYPE Type\*

### 错误处理相关全局变量

1. **extern** **int** yylineno; //当前行号
2. **extern** **char**\* yytext; //词法分析当前缓冲区（单词）的内容
3. **extern** **char** lineBuffer[500]; //当前行的所有内容
4. **extern** **int** yyleng; //当前匹配字符串的长度
5. **extern** **int** yycolumn;  //当前列号
6. **extern** string itos(**int** num);  //整型转string型函数
7. //以上全局变量来自于词法分析，故有extern关键字
9. **bool** haveSemanticError=**false**; //是否有语法错误
10. vector<string> syntaxErrorInformation; //存放语法错误信息
12. Type\* ParseTreeHead=NULL; //整棵语法分析树的根节点

### 错误处理相关函数声明

1. **void** yyerror(**const** **char** \*s, YYLTYPE \*loc); //处理单个文法符号的错误信息
2. **void** yyerror(**const** **char** \*s, **int** line, **int** col); //处理位于一行以内的错误信息
3. **void** yyerror(**const** **char** \*s, **int** startLine, **int** startCol, **int** endLine, **int** endCol); //处理涉及到多行的错误信息

由此，加上Bison(YACC)自带的void yyerror(const char \*s)用于报告致命性（终止型）语法错误，我们采用重载的机制，共有四类错误信息处理函数，应对不同的情况。

## 构建语法树过程描述

我们知道：

一个文法*G*是下述元素构成的一个四元组（*N*, Σ,*P*,*S*）：

* “**非终结符号**”集合*N*。
* “**终结符号**”集合Σ，Σ与*N*无交。
* 取如下形式的一组“**产生式规则**”*P*，
* (Σ ∪*N*)\*中的字符串→ (Σ ∪*N*)\* 中的字符串，并且产生式左侧的字符串中必须至少包括一个非终结符号。
* “**起始符号**”*S*，*S*属于*N*。

故语法树构建过程的设计如下流程图所示：

以上这三个步骤在我们编写的文法文件（Bison Grammar File）有清晰的展示。

* 非终结符的定义：加上%token的前缀。

1. %token PROGRAM
2. %token CONST
3. %token VAR
4. %token ARRAY
5. %token OF
6. %token PROCEDURE
7. %token FUNCTION
8. %token \_BEGIN
9. %token END
10. …………

* 起始符的定义：加上%start的前缀

1. %start programstruct

* 构造结点和子树（即后两个步骤）

下面以idlist的产生式为例，说明结点和子树的改造过程：

1. idlist: idlist ',' IDENTIFIER{
2. $$=**new** Type;
3. $$->token = "idlist";
4. $$->children.push\_back($1); $$->children.push\_back($2); $$->children.push\_back($3);
5. }|IDENTIFIER{
6. $$=**new** Type;
7. $$->token = "idlist";
8. $$->children.push\_back($1);
9. };

$$表示产生式左部符号，$1、$2、$3……表示产生式右部第一个、第二个、第三个……符号。

## 错误处理相关设计

### 错误处理种类

1. **程序识别失败**，遇到该错误时报错并终止语法分析程序
2. **出现非法记号**，遇到该错误时报错并跳过该非法记号继续分析。
3. **非终结符号识别失败，**如不完整的函数头、过程头，不完整的参数列表，不完整的数组下表列表，错误的关键字等，遇到该错误时报错并继续分析。
4. **标点符号缺失**，遇到该错误时报错并继续分析。
5. **运算符号缺失**，遇到该错误时报错并继续分析。
6. **常数初始化右值缺失**，遇到该错误时报错并继续分析。

### 错误处理相关产生式处理

在YACC工具中，采用引入错误产生式的方法，即在出错的位置插入一个名为error的token到输入中，并在相应翻译方案中调用yyerror错误处理函数。错误产生式的翻译方案中建立残缺的语法树，只将产生式左部符号的节点加入到语法树中，右部的文法符号抛弃。

以id\_varpart的产生式为例：

1. id\_varpart: '[' expression\_list ']'{ //正常
2. $$=**new** Type;
3. $$->token="id\_varpart";
4. $$->children.push\_back($1);$$->children.push\_back($2);$$->children.push\_back($3);
5. }|'[' error{ //ERROR 不完整的数组下标列表 checked
6. $$=**new** Type;
7. $$->token="id\_varpart";
8. yyerror("incomplete expression list of array subindex", &@$);
9. }|'[' expression\_list error{ //ERROR 缺失右中括号 checked
10. $$=**new** Type;
11. $$->token="id\_varpart";
12. yyerror("missing a right square bracket here", @2.last\_line, @2.last\_column+1);
13. }|{ //正常
14. $$=**new** Type;
15. $$->token="id\_varpart";
16. };

error代替了正确文法符号的位置，这里有两种错误情况，第一种是不完整的数组下标列表，第二种是右括号缺失，分别对应的是处理单个符号的错误信息和处理一行内的错误信息。

在词法分析详细设计中已经提到YYLTYPE的数据结构，first\_line表示当前文法符号代表的内容的起始行，first\_column表示当前文法符号代表的内容的起始列，last\_line表示当前文法符号代表的内容的终止行，last\_column表示当前文法符号代表的内容的终止列。

### 错误处理相关函数设计

除YACC自带的yyerror函数用于报告致命性（终止型）语法错误以外，我们使用重载设计了另外三种错误处理机制：

1. **void** yyerror(**const** **char** \*s){
2. haveSemanticError = **true**;//错误标志，含有语法错误
3. string errorInformation;//定义错误信息
4. errorInformation += string(s);//添加错误信息
5. errorInformation += ", location: " + itos(yylineno-1) + "." + itos(yycolumn-yyleng);//添加错误位置
6. syntaxErrorInformation.push\_back(errorInformation);//存放错误信息
7. }
9. **void** yyerror(**const** **char** \*s, YYLTYPE \*loc){//处理单个字符的错误
10. haveSemanticError = **true**;
11. string errorInformation;
12. errorInformation = "syntax error, " + string(s) + ", location: " + itos(loc->first\_line) + "." + itos(loc->first\_column) + "-" + itos(loc->last\_line) + "." + itos(loc->last\_column);
13. syntaxErrorInformation.push\_back(errorInformation);
14. }
16. **void** yyerror(**const** **char** \*s, **int** line, **int** col){//处理一行以内的错误
17. haveSemanticError = **true**;
18. string errorInformation;
19. errorInformation = "syntax error, " + string(s) + ", location: " + itos(line) + "." + itos(col);
20. syntaxErrorInformation.push\_back(errorInformation);
21. }
23. **void** yyerror(**const** **char** \*s, **int** startLine, **int** startCol, **int** endLine, **int** endCol){//处理涉及多行的错误
24. haveSemanticError = **true**;
25. string errorInformation;
26. errorInformation = "syntax error, " + string(s) + ", location: " + itos(startLine) + "." + itos(startCol) + "-" + itos(endLine) + "." + itos(endCol);
27. syntaxErrorInformation.push\_back(errorInformation);
28. }

错误信息格式为：

syntax error +错误内容+location：错误内容起始位置所在行.起始位置所在列-终止位置所在行.终止位置所在列（若是一行内单个符号出现错误，只需报告该符号的所在行.所在列）。

将所有错误信息存入 syntaxErrorInformation中，便于语法分析结束后将所有的语法错误一并输出。

# 普通语法分析树转AST详细设计

为了便于后续进行语义分析和代码生成，需要去除语法分析树中的冗余部分，将其转化为一棵只包含有用信息的抽象语法树（AST），抽象语法树的设计参见概要设计{loc}。

该部分首先介绍AST的节点类定义，然后介绍设计的函数接口，随后以转移图的形式介绍函数之间的调用关系，最后以伪代码的形式给出所有函数的详细设计实现。

**往下读就可以观察到，在转化为AST的函数中，出现了许多命名相似的函数。例如getValueParameter、getParameter、getFormalParameter、getFormalParaList，又例如setConst、getConst、getConstList。出现这样的命名情况并不是我们的命名习惯问题，而正是普通语法分析树的冗余信息问题导致的。当然我们也可以写出一长串的指针链，来避免这种函数命名情况，但是为了编程的方便，拒绝长串指针链（而且长指针链更容易写出BUG），我们采用了这种看似复杂的命名方法。接下来我们会尽量讲清楚这些函数各自的用途和相互之间的调用关系。**

**想要深入理解整个普通语法树转AST的过程，需要通读下面的文档，在粗略了解了接口之后，需要再深入阅读调用关系和各函数的具体实现。**

**需要时刻牢记于心的是，Type\* 指针提供的是普通语法树的节点，提供的是输入信息；其它的所有类指针提供的是AST的节点，提供的是输出信息的载体。**

## AST节点类定义

**抽象语法树在概要设计中已经给出了详细设计。**

### 节点类定义代码

class \_Expression //表达式

{

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

string operation;//具体操作符

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

\_Expression \*operand1,\*operand2;//操作数

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

};

class \_FunctionCall //函数调用

{

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

};

class \_VariantReference //变量引用

{

public:

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

int flag;//0表示非数组,1表示数组

vector<\_Expression\*> expressionList;//各维引用表达式列表

};

class \_Statement //语句的基类

{

public:

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

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

        int lineNumber;//行号

};

class \_Compound:public \_Statement //复合语句

{

public:

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

        //行号由begin的位置决定

};

class \_RepeatStatement:public \_Statement //repeat循环语句

{

public:

\_Expression \*condition;//条件表达式

\_Statement \*\_do;//循环体语句

        //行号由repeat的位置决定

};

class \_WhileStatement:public \_Statement //while循环语句

{

public:

\_Expression \*condition;//条件表达式

\_Statement \*\_do;//循环体语句

        //行号由while的位置决定

};

class \_ForStatement:public \_Statement //for循环语句

{

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

};

class \_IfStatement:public \_Statement //if条件语句

{

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

};

class \_AssignStatement:public \_Statement //赋值语句

{

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

};

class \_ProcedureCall:public \_Statement //过程调用

{

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

};

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

};

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，形式参数一定是基本类型，所以改为string

int flag;//flag=0表示传值调用，flag=1表示引用调用

};

class \_FunctionDefinition //函数或过程定义

{

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<\_Variant\*> variantList;//变量定义列表

\_Compound\* compound;

};

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

};

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"id","integer","real","char","boolean"

pair<string,int> valueId;

char charValue;

        int intValue;

float realValue;

        string strOfVal;//所有常量取值的字符串表示

        bool isMinusShow;

};

class \_SubProgram //分程序

{

public:

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

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

};

class \_Program //程序

{

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

};

### 节点类定义细节

#### \_Expression的种类设计

在原来的文法中，按照运算符的优先级定义了mulop、relop、addop，然后又按照这个优先级的嵌套顺序，将表达式细化为了expression、simple expression、term、factor这四类，从前往后，从复杂到简单。由于我们只定义了\_Expression这一个节点类来表示表达式，所以势必得加入一些域，以区别这些不同的情况。

首先，我们引入了type域，取值情况如下:

|  |  |
| --- | --- |
| type取值 | 描述 |
| var | 变量引用，是一个标识符，或者标识符+中括号包围的下标列表。再具体分，则可能是变量、常量、数组元素、函数名等（这个分类将在语义分析中考虑，语法分析不考虑） |
| integer | 无符号整数 |
| real | 浮点数 |
| char | 字符常量 |
| function | 函数调用 |
| compound | 包含运算符的复杂表达式 |

compound表示包含运算符的复杂表达式，又可以细分为很多种情况，所以用引入operationType表示运算符类型，operation表示具体的运算符是什么，取值如下：

|  |  |  |
| --- | --- | --- |
| operationType取值 | operation取值 | 描述 |
| relop | > | 大于 |
| >= | 大于等于 |
| < | 小于 |
| <= | 小于等于 |
| <> | 不等于 |
| = | 等于 |
| mulop | \* | 乘 |
| / | 除 |
| div | 整除 |
| mod | 取余 |
| and | 且 |
| addop | + | 加 |
| - | 减 |
| or | 或 |
| single | minus | 取相反数 |
| not | 非 |
| bracket | 在表达式两端添加括号 |

#### \_VariantRerence区分是否数组

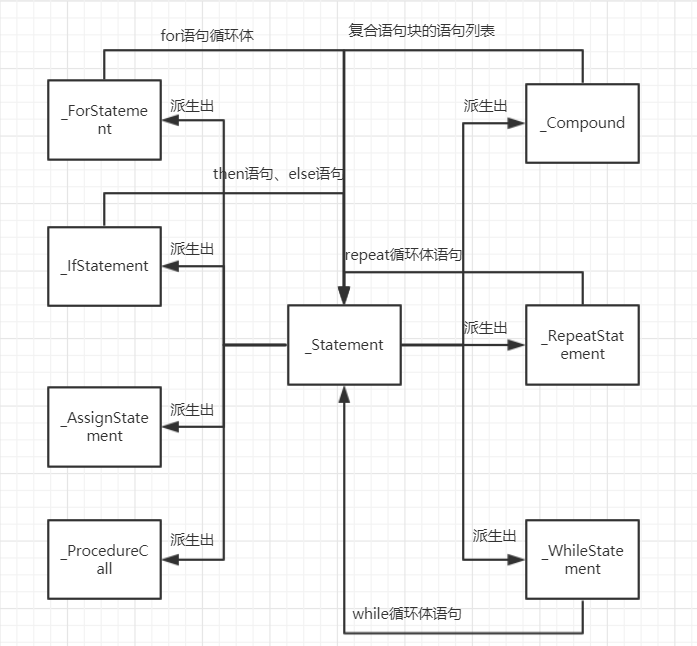
引入flag，若flag=0，则表示不带中括号包围的下标表达式，若flag=1，则表示带中括号包围的下标表达式，即数组元素的引用。

#### \_Statement类之间的关系

首先\_Statement是基类，\_Compound、\_RepeatStatement、\_WhileStatement、\_ForStatement、\_IfStatement、\_AssignStatement、\_ProcedureCall均为\_Statement的派生类。所有派生类的域中，涉及到语句的，均用\_Statement基类的指针来表示，因为在生成AST之前并不知道具体是哪个语句。

例如\_Compound类的语句列表是\_Statement类指针的列表，再例如\_WhileStatement循环体语句也是\_Statement类指针。

这些类之间的关系可以用下面的关系图来表示。



我们在具体代码实现时，如果要用到派生类的变量或方法，需要强制类型转化为派生类，因此需要引入一个type域，表示当前语句实际上是什么派生类。

|  |  |
| --- | --- |
| 派生类 | type取值 |
| \_Compound | compound |
| \_RepeatStatement | repeat |
| \_WhileStatement | while |
| \_ForStatement | for |
| \_IfStatement | if |
| \_AssignStatement | assign |
| \_ProcedureCall | procedure |

#### \_Type区分是否数组

引入flag，若flag=0，表示基本类型，若flag=1，表示数组类型。

#### \_FormalParameter区分是否是引用参数

引入flag，若flag=0，表示传值参数，若flag=1，表示引用参数

#### \_Constant常量定义细节

PASCAL-S在定义常量时，首先赋值号用的是等号，其次也不会指定类型，所以常量的类型需要我们自己根据读取到的常量进行判断；同时也支持用已定义的常量的取值来初始化当前定义的常量；且不支持bool类型常量。

综上，我们需要引入一个type表示常量定义的类型：

|  |  |
| --- | --- |
| type取值 | 描述 |
| id | 由别的常量取值初始化 |
| integer | 由整数初始化 |
| real | 由浮点数初始化 |
| char | 由字符常量初始化 |

另外，在定义时，还可以在右值前添加正负号，添加正号与不添加正负号无异，添加负号表示需要取相反数，因此引入bool类型变量isMinusShow表示常量是否取相反数。注意，isMinusShow不能表示是否是负数，当常量取值由别的常量定义时，别的常量可能本身已经是负数，再添加一个负号，就得到了一个正数。

#### 不同节点类的类型区别

有一些语法成分的类型只能是基本类型，而有一些语法成分的类型还可以是数组，具体如下表所示：

|  |  |
| --- | --- |
| 语法成分 | 类型 |
| 形式参数 | 基本类型 |
| 函数返回值 | 基本类型 |
| 变量 | 基本类型或数组 |
| 常量 | 除了boolean之外的基本类型 |

## 函数接口设计

### 辅助函数

这一部分的函数为转化为AST的过程提供辅助。

#### 字符串转整型

* 函数接口

int str2int(string str);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string str | 需要转化的字符串 |

* 返回值

int，表示转化后的整型数字

#### 字符串转浮点型

* 函数接口

float str2float(string str);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| string str | 需要转化的字符串 |

* 返回值

float，表示转化后的浮点型数字

### 转化为AST的函数

这一部分的函数基本上与PASCAL-S的文法中的文法符号一一对应，通过相互之间的（递归）调用，最终将普通的语法分析树转化为抽象语法树。

#### 获取标识符列表

* 函数接口

void getIdList(Type \*now, vector< pair<string, int> >& res,bool reverseFlag);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector< pair<string, int> >& res | 存放标识符及其所在行号的列表 |
| bool reverseFlag | 是否需要翻转标识符列表 |

* 返回值

无

#### 获取数组上下界列表

* 函数接口

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

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector< pair<int, int> >& \_arrayRangeList | 数组上下界列表 |

* 返回值

无

#### 获取类型

* 函数接口

\_Type\* getType(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Type\*，表示类型

#### 获取变量定义

* 函数接口

void getVariant(Type \*now, vector<\_Variant\*>& \_variantList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Variant\*>& \_variantList | 变量节点列表，新获取的变量节点需添加到该变量节点列表中 |

* 返回值

无

#### 获取变量定义列表

* 函数接口

void getVariantList(Type \*now, vector<\_Variant\*>& \_variantList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Variant\*>& \_variantList | 变量节点列表 |

* 返回值

无

#### 获取常量定义节点具体信息

* 函数接口

void setConst(Type \*now, \_Constant\* &\_constant);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| \_Constant\* &\_constant | 将获取的常量信息保存到该常量节点类中 |

* 返回值

无

#### 获取常量定义节点

* 函数接口

void getConst(Type \*now, vector<\_Constant\*>& \_constantList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Constant\*>& \_constantList | 常量定义节点列表，新获取的常量定义节点需要添加到该列表中 |

* 返回值

无

#### 获取常量定义节点列表

* 函数接口

void getConstList(Type \*now, vector<\_Constant\*>& \_constantList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Constant\*>& \_constantList | 常量定义节点列表 |

* 返回值

无

#### 获取形式参数的具体信息

* 函数接口

void getValueParameter(Type \*now, vector<\_FormalParameter\*>& \_formalParaList, int flag = 0);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_FormalParameter\*>& \_formalParaList | 形式参数节点列表，新获取的形式参数节点需添加到该列表中 |
| int flag | flag=0表示传值参数，flag!=0表示传引用参数 |

* 返回值

无

* 备注

该函数获取的不一定是单纯的传值参数，该函数的参数flag就是为了表示当前语法分析树节点表示的是单纯的传值参数，还是去除了VAR关键字的传引用参数。

#### 区别形式参数

* 函数接口

void getParameter(Type \*now, vector<\_FormalParameter\*>& \_formalParaList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_FormalParameter\*>& \_formalParaList | 形式参数节点列表 |

* 返回值

无

* 备注

该函数用于区别所识别的是传值参数还是传引用参数，并以此来决定调用getValueParameter时flag的取值

#### 获取形式参数

* 函数接口

void getFormalParameter(Type \*now, vector<\_FormalParameter\*>& \_formalParaList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_FormalParameter\*>& \_formalParaList | 形式参数节点列表 |

* 返回值

无

#### 获取形式参数节点列表

* 函数接口

void getFormalParaList(Type \*now, vector<\_FormalParameter\*>& \_formalParaList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_FormalParameter\*>& \_formalParaList | 形式参数节点列表 |

* 返回值

无

#### 获取子程序头

* 函数接口

void getSubprogramHead(Type \*now, pair<string, int>& functionID, vector<\_FormalParameter\*>& \_formalParaList, pair<string,int> &\_type);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| pair<string, int>& functionID | 子程序名及行号 |
| vector<\_FormalParameter\*>& \_formalParaList | 形式参数节点列表 |
| 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

无

#### 获取factor

* 函数接口

\_Expression\* getFactor(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Expression\*，指向表达式节点类

#### 获取term

* 函数接口

\_Expression\* getTerm(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Expression\*，指向表达式节点类

#### 获取simple expression

* 函数接口

\_Expression\* getSimpleExpression(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Expression\*，指向表达式节点类

#### 获取expression

* 函数接口

\_Expression\* getExpression(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Expression\*，指向表达式节点类

#### 获取表达式列表

* 函数接口

void getExpressionList(Type \*now, vector<\_Expression\*>& \_expressionList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Expression\*>& \_expressionList | 表达式节点列表 |

* 返回值

无

#### 获取变量引用节点

* 函数接口

\_VariantReference\* getVariantReference(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_VariantReference\*，指向变量引用节点

#### 获取变量引用节点列表

* 函数接口

void getVariantReferenceList(Type \*now, vector<\_Expression\*>& \_expressionList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Expression\*>& \_expressionList | 表达式节点类列表 |

* 返回值

#### 获取else部分语句

* 函数接口

\_Statement\* getElseStatement(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Statement\*，指向语句节点类

#### 获取过程调用

* 函数接口

\_Statement\* getProcedureCall(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Statement\*，指向语句节点类

#### 获取各种类型的语句

* 函数接口

\_Statement\* getStatement(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Statement\*，指向语句节点类

#### 获取语句列表

* 函数接口

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

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Statement\*>& \_statementList | 语句节点类列表 |

* 返回值

无

#### 获取复合语句块

* 函数接口

\_Compound\* getCompoundStatement(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Compound\*，指向复合语句块节点类

#### 获取子程序体

* 函数接口

void getSubprogramBody(Type \*now, vector<\_Constant\*>& \_constList, vector<\_Variant\*>& \_variantList, \_Compound\* &\_compound);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_Constant\*>& \_constList | 常量定义节点类列表 |
| vector<\_Variant\*>& \_variantList | 变量定义节点类列表 |
| \_Compound\* &\_compound | 复合语句块 |

* 返回值

无

#### 获取函数定义

* 函数接口

\_FunctionDefinition\* getSubprogramDefinition(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_FunctionDefinition\*，指向函数定义节点类

#### 获取子程序定义列表

* 函数接口

void getSubprogramDefinitionList(Type \*now, vector<\_FunctionDefinition\*>& \_subprogramDefinitionList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| vector<\_FunctionDefinition\*>& \_subprogramDefinitionList | 子程序定义节点类列表 |

* 返回值

无

#### 获取主程序体

* 函数接口

\_SubProgram\* getProgramBody(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_SubProgram\*，指向分程序节点类

#### 获取主程序头

* 函数接口

void getProgramHead(Type \*now, pair<string, int>& \_programId, vector< pair<string, int> >& \_paraList);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |
| pair<string, int>& \_programId | 主程序名及其行号 |
| vector< pair<string, int> >& \_paraList | 主程序参数名及其行号列表 |

* 返回值

无

#### 获取整个程序

* 函数接口

\_Program\* getProgram(Type \*now);

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| Type \*now | 当前语法分析树节点 |

* 返回值

\_Program\*，抽象语法树的根节点指针

## 各函数伪代码

### 辅助函数

#### 字符串转整型

int str2int(string str){

int res=0;

int len=str.length();

for(int i=0;i<len;i++){

res=res\*10;

res+=str[i]-'0';

}

return res;

}

#### 字符串转浮点型

float str2float(string str){

float res=0;

int len=str.length();

int loc=str.find('.');

for(int i=0;i<loc;i++){

res\*=10;

res+=str[i]-'0';

}

float base=1;

for(int i=loc+1;i<len;i++){

base=base\*0.1;

res+=base\*(str[i]-'0');

}

return res;

}

### 转化为AST的函数

这一部分函数在实现的时候，必须参照文法的设计，才能搞清楚普通语法分析树每个节点是什么，每个节点有哪些孩子，这些孩子之间按照什么顺序排列。

且为了方便DEBUG，检测程序编写错误，每个函数在一开始都要检查当前的普通语法分析树节点是否是需要的那类节点。例如getIdList函数需要检查Type\*节点的记号是否是”idlist”。

#### 获取标识符列表

void getIdList(Type \*now,vector< pair<string,int> >& res,bool reverseFlag){

if(now->token!="idlist"){

cout << "getIdList error" << endl;

return;

}

if(是产生式（4.1）){

res中加入当前id及其行号;

getIdList(now->children[0],res,reverseFlag);//递归调用获取下一个id

}

else{//产生式（4.2）

res中加入当前id及其行号;

        if(reverseFlag)

            res反转;

}

}

#### 获取数组上下界列表

void getArrayRangeList(Type \*now,vector< pair<int,int> >& \_arrayRangeList){

if(now->token!="period"){

cout << "getArrayRangeList error" << endl;

return;

}

\_arrayRangeList中加入当前维上下界;

if(是产生式（11.1）)

递归调用getArrayRangeList获取数组的下一维上下界;

else//产生式（11.2）

\_arrayRangeList反转;

}

#### 获取类型

\_Type\* getType(Type \*now){

if(now->token!="type"){

cout << "getType error" << endl;

return NULL;

}

\_Type\* \_type = new \_Type;//新建一个\_Type节点

\_type中加入基本类型及其行号;

if(是产生式10.2){

\_type标记为数组类型;

调用getArrayRangeList获取数组上下界

}

else//产生式10.1

\_type标记为非数组类型;

return \_type;

}

#### 获取变量定义

void getVariant(Type \*now,vector<\_Variant\*>& \_variantList){

if(now->token!="var\_declaration"){

cout << "getVariant error" << endl;

return;

}

vector< pair<string,int> > \_idList;

调用getIdList获取\_idList;//不反转

\_Type \*\_type=调用getType获取变量类型;

遍历\_idList

新建变量节点，加入id和类型信息，并添加到\_variantList中;

if(是产生式9.1)

调用getVariant获取下一段变量定义;

else//产生式9.2

\_variantList反转;

}

#### 获取变量定义列表

void getVariantList(Type \*now,vector<\_Variant\*>& \_variantList){

if(now->token!="var\_declarations"){

cout << "getVariantList error" << endl;

return;

}

if(是产生式8.1)

调用getVariant获取变量定义列表;

}

#### 获取常量定义节点具体信息

void setConst(Type \*now,\_Constant\* &\_constant){//pascal在定义常量时，并没有指定常量的类型，所以需要自行判断

if(now->token!="const\_value"){

cout << "setConst error" << endl;

return;

}

if(是产生式7.1或7.2或7.3){

\_constant类型标记为"id"，即由别的常量标识符定义其取值;

\_constant添加id及其行号;

        \_constant添加定义其取值的常量标识符;

if(是产生式7.2)

         \_constant标记出现了负号;

}

else if(是产生式7.4或7.5或7.6){

\_constant类型标记为"integer";

\_constant->intValue=str2int(字符串表示的无符号整数);

        \_constant->strOfVal=字符串表示的无符号整数;

if(是产生式7.5)

         \_constant标记出现了负号;

}

else if(是产生式7.7或7.8或7.9){

\_constant类型标记为"real";

\_constant->realValue=str2float(字符串表示的无符号浮点数);

        \_constant->strOfVal=字符串表示的无符号浮点数;

if(是产生式7.8)

         \_constant标记出现了负号;

}

else if(是产生式7.10){

\_constant类型标记为"char";

\_constant->charValue=字符常量;

        \_constant->strOfVal=now->字符常量;

}

else{

cout << "setConst error" << endl;

}

}

#### 获取常量定义节点

void getConst(Type \*now,vector<\_Constant\*>& \_constantList){

if(now->token!="const\_declaration"){

cout << "getConst error" << endl;

return;

}

\_Constant\* \_constant=new \_Constant;

\_constant添加id及其行号;

调用setConst获取\_constant的剩余信息;

\_constant添加到\_constantList中;

if(是产生式6.1)

调用getConst获取下一个常量定义;

else

\_constantList反转;

}

#### 获取常量定义节点列表

void getConstList(Type \*now,vector<\_Constant\*>& \_constantList){

if(now->token!="const\_declarations"){

cout << "getConstList error" << endl;

return;

}

if(是产生式5.1)

调用getConst获取常量定义列表;

}

#### 获取形式参数的具体信息

void getValueParameter(Type \*now,vector<\_FormalParameter\*>& \_formalParaList,int flag){

if(now->token!="value\_parameter"){

cout << "getValueParameter error" << endl;

return;

}

vector< pair<string,int> > \_idList;

调用getIdList获取\_idList;//不反转

遍历idList

新建一个形参节点类，添加id，类型，传值参数还是引用参数等信息，并添加到\_formalParaList中;

}

#### 区别形式参数

void getParameter(Type \*now,vector<\_FormalParameter\*>& \_formalParaList){

if(now->token!="parameter"){

cout << "getParameter error" << endl;

return;

}

if(是产生式17.1)//引用参数

调用getValueParameter获取引用参数列表，调用时表明是引用参数;

else if(是产生式17.2)//传值参数

调用getValueParameter获取传值参数列表，调用时表明是传值参数;

else

cout << "getParameter error" << endl;

}

#### 获取形式参数

void getFormalParameter(Type \*now,vector<\_FormalParameter\*>& \_formalParaList){

if(now->token!="parameter\_list"){

cout << "getFormalParameter error" << endl;

return;

}

调用getParameter获取当前这一段形式参数列表;

if(是产生式16.1)

调用getFormalParameter获取下一段形式参数列表;

else//产生式16.2

\_formalParaList反转;

}

#### 获取形式参数节点列表

void getFormalParaList(Type \*now,vector<\_FormalParameter\*>& \_formalParaList){

if(now->token!="formal\_parameter"){

cout << "getFormalParaList error" << endl;

return;

}

if(是产生式15.1)

调用getFormalParameter形式参数列表;

}

#### 获取子程序头

void getSubprogramHead(Type \*now,pair<string,int>& functionID,vector<\_FormalParameter\*>& \_formalParaList,pair<string,int> &\_type){

if(now->token!="subprogram\_head"){

cout << "getSubprogramHead error" << endl;

return;

}

获取子程序名及其行号赋值给functionID;

调用getFormalParaList获取形式参数列表;

if (是产生式14.2) //表示函数

        \_type=获取返回值类型及其行号;

else //产生式14.1 表示过程

     \_type=空;//表示没有返回值

}

#### 获取factor

//"var"表示变量,"integer"表示整数,"real"表示浮点数,"char"表示字符常量

//"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

\_Expression\* getFactor(Type \*now){

if(now->token!="factor"){

cout << "getFactor error" << endl;

        return NULL;

}

\_Expression\* \_expression = new \_Expression;

if(是产生式32.1){

\_expression类型标记为"integer";

        \_expression->strOfNum=字符串表示的无符号整数;

\_expression->intNum=str2int(字符串表示的无符号整数);

所在行号=整数所在行号;

}

else if(是产生式32.2){

\_expression类型标记为"real";

        \_expression->strOfNum=字符串表示的无符号浮点数;

\_expression->realNum=str2int(字符串表示的无符号浮点数);

所在行号=浮点数所在行号;

}

else if(是产生式32.3){

\_expression类型标记为"var";

调用getVariantReference获取变量引用节点;

        所在行号=变量引用所在行号;

}

else if(是产生式32.4){

\_expression类型标记为"function";

新建一个\_FunctionCall节点;

获取函数名及其行号;

调用getExpressionList获取函数调用的实参表达式列表;

        所在行号=函数名所在行号;

}

else if(是产生式32.5){

\_expression类型标记为"compound";

运算符="bracket";//在表达式两边添加括号

        运算符类型="single";//单目运算符

调用getExpression获取操作数表达式;

        所在行号=操作数表达式所在行号;

}

else if(是产生式32.6){

\_expression类型标记为"compound";

运算符="not";//取非

        运算符类型="single";//单目运算符

调用getFactor获取操作数表达式;

        所在行号=操作数表达式所在行号;

}

else if(是产生式32.7){

\_expression类型标记为"compound";

运算符="minus";//取相反数

        运算符类型="single"; //单目运算符

调用getFactor获取操作数表达式;

        所在行号=操作数表达式所在行号;

}

    else if (是产生式32.8) {

        \_expression类型标记为"char";

        \_expression->charVal=字符常量;

        所在行号=字符常量所在行号;

    }

else{

cout << "getFactor error" << endl;

return NULL;

}

return \_expression;

}

#### 获取term

\_Expression\* getTerm(Type \*now){

if(now->token!="term"){

cout << "term" << endl;

        return NULL;

}

\_Expression\* \_expression=NULL;

if(是产生式31.1){

\_expression = new \_Expression;

\_expression类型标记为"compound";

获取运算符;

运算符类型="mulop";

调用getTerm获取左操作数表达式;

调用getFactor获取右操作数表达式;

        所在行号=左操作数表达式所在行号;

}

else //是产生式31.2

\_expression=调用getFactor获取表达式;

return \_expression;

}

#### 获取simple expression

\_Expression\* getSimpleExpression(Type \*now){

if(now->token!="simple\_expression"){

cout << "getSimpleExpression error" << endl;

        return NULL;

}

\_Expression\* \_expression=NULL;

if(是产生式30.1){

\_expression = new \_Expression;

\_expression类型标记为"compound";

获取运算符;

运算符类型="addop";

调用getSimpleExpression获取左操作数表达式;

调用getTerm获取右操作数表达式;

        所在行号=左操作数表达式所在行号;

}

else //是产生式30.2

\_expression=调用getTerm获取表达式;

return \_expression;

}

#### 获取expression

\_Expression\* getExpression(Type \*now){

if(now->token!="expression"){

cout << "getExpression error" << endl;

        return NULL;

}

\_Expression\* \_expression=NULL;

if(是产生式29.1或29.2){

\_expression = new \_Expression;

\_expression类型标记为="compound";

获取运算符;

运算符类型="relop";

调用getSimpleExpression获取左操作数表达式;

调用getSimpleExpression获取右操作数表达式;

        \_expression->lineNumber=左操作数表达式所在行号

}

else

\_expression=getSimpleExpression(now->children[0]);

return \_expression;

}

#### 获取表达式列表

void getExpressionList(Type \*now,vector<\_Expression\*>& \_expressionList){

if(now->token!="expression\_list"){

cout << "getExpressionList error" << endl;

return;

}

获取当前表达式，添加到\_expressionList中;

if(是产生式28.1)

调用getExpressionList获取下一个表达式;

else//产生式28.2

\_expressionList反转;

}

#### 获取变量引用节点

void getVariantReferenceList(Type \*now,vector<\_Expression\*>& \_expressionList){

if(now->token!="id\_varpart"){

cout << "getVariantReferenceList error" << endl;

return;

}

if(是产生式26.1)

调用getExpressionList获取数组下标表达式列表;

}

#### 获取变量引用节点列表

\_VariantReference\* getVariantReference(Type \*now){

if(now->token!="variable"){

cout << "getVariantReference error" << endl;

        return NULL;

}

\_VariantReference\* \_variantReference = new \_VariantReference;

获取id及其行号;

调用getVariantReferenceList获取数组下标表达式列表;

    if (下标表达式列表不为空)

        标记为数组;

    else

        标记为非数组;

return \_variantReference;

}

#### 获取else部分语句

\_Statement\* getElseStatement(Type \*now){

if(now->token!="else\_part"){

cout << "getElseStatement error" << endl;

return NULL;

}

if(是产生式24.2)

return NULL;

//产生式24.1

return 调用getStatement获取else部分语句;

}

#### 获取过程调用

\_Statement\* getProcedureCall(Type \*now) {

    if (now->token != "procedure\_call") {

        cout << "getProcedureCall error" << endl;

        return NULL;

    }

    \_ProcedureCall \*\_procedureCall = new \_ProcedureCall;

    获取过程调用所在行号;

    \_procedureCall类型标记为"procedure";

    获取过程id及其行号;

    if (是产生式27.2)

        调用getExpressionList获取过程调用实参表达式列表;

    return \_procedureCall;

}

#### 获取各种类型的语句

\_Statement\* getStatement(Type \*now){

if(now->token!="statement"){

cout << "getStatement error" << endl;

return NULL;

}

    if (是产生式23.8)

        return NULL;

if(是产生式23.1){

\_AssignStatement \*\_assignStatement = new \_AssignStatement;

        获取赋值语句所在行号;

\_assignStatement类型标记为"assign";

获取左值变量引用;

获取右值表达式;

return \_assignStatement;

}

else if(是产生式23.2)

        return 调用getProcedureCall获取过程调用语句;

else if(是产生式23.3)

        return 调用getCompoundStatement获取复合语句块;

else if(是产生式23.4){

\_IfStatement\* \_ifStatement = new \_IfStatement;

        获取if语句所在行号;

\_ifStatement类型标记为"if";

调用getExpression获取condition表达式;

调用getStatement获取then语句;

调用getElseStatement获取else语句;

return \_ifStatement;

}

else if(是产生式23.5){

\_ForStatement\* \_forStatement = new \_ForStatement;

        获取for语句所在行号;

\_forStatement类型标记为"for";

获取循环变量id;

调用getExpression获取初值表达式;

调用getExpression获取终值表达式;

调用getStatement获取循环体语句;

return \_forStatement;

}

else if(是产生式23.6){

\_WhileStatement\* \_whileStatement = new \_WhileStatement;

        获取while语句所在行号;

\_whileStatement类型标记为"while";

调用getExpression获取condition表达式;

调用getStatement获取循环体语句;

return \_whileStatement;

}

else if(是产生式23.7){

\_RepeatStatement\* \_repeatStatement = new \_RepeatStatement;

        获取repeat语句所在行号;

\_repeatStatement类型标记为"repeat";

调用getExpression获取condition表达式;

调用getStatement获取循环体语句;

return \_repeatStatement;

}

else{

cout << "[getStatement] statement token error" << endl;

return NULL;

}

}

#### 获取语句列表

void getStatementList(Type \*now,vector<\_Statement\*>& \_statementList){

if(now->token!="statement\_list"){

cout << "getStatementList error" << endl;

return;

}

    调用getStatement获取当前语句;

    if(获取的语句不为空)

        将获取的语句添加到\_statementList中;

if(是产生式22.1)

调用getStatementList获取下一条语句;

else

\_statementList列表翻转;

}

#### 获取复合语句块

\_Compound\* getCompoundStatement(Type \*now){

if(now->token!="compound\_statement"){

cout << "getCompoundStatement error" << endl;

return NULL;

}

    \_Compound \*\_compound = new \_Compound;

    获取compound语句所在行号;

    \_compound类型标记为"compound";

调用getStatementList获取复合语句块的语句列表;

return \_compound;

}

#### 获取子程序体

void getSubprogramBody(Type \*now,vector<\_Constant\*>& \_constList,vector<\_Variant\*>& \_variantList,\_Compound\* &\_compound){

if(now->token!="subprogram\_body"){

cout << "getSubprogramBody error" <<endl;

return;

}

调用getConstList获取常量定义列表;

调用getVariantList获取变量定义列表;

    调用getCompoundStatement复合语句块;

}

#### 获取函数定义

\_FunctionDefinition\* getSubprogramDefinition(Type \*now){

if(now->token!="subprogram"){

cout << "getSubprogramDefinition error" << endl;

return NULL;

}

\_FunctionDefinition \*\_functionDefinition=new \_FunctionDefinition;

调用getSubprogramHead获取子程序定义头;

调用getSubprogramBody获取子程序定义体;

return \_functionDefinition;

}

#### 获取子程序定义列表

void getSubprogramDefinitionList(Type \*now,vector<\_FunctionDefinition\*>& \_subprogramDefinitionList){

if(now->token!="subprogram\_declarations"){

cout << "getSubprogramDefinitionList error" << endl;

return;

}

    if (是产生式12.1) {

调用getSubprogramDefinition获取子程序定义,并添加到子程序定义列表中;

        调用getSubprogramDefinitionList获取下一个子程序定义;

    }

else //产生式12.2

\_subprogramDefinitionList反转;

}

#### 获取主程序体

\_SubProgram\* getProgramBody(Type \*now){

if(now->token!="program\_body"){

cout << "getProgramBody error" << endl;

return NULL;

}

\_SubProgram \*\_subProgram=new \_SubProgram;

调用getConstList获取常量定义列表;

调用getVariantList获取变量定义列表;

调用getSubprogramDefinitionList获取子程序定义列表;

    调用getCompoundStatement获取主程序体;

return \_subProgram;

}

#### 获取主程序头

void getProgramHead(Type \*now,pair<string,int>& \_programId,vector< pair<string,int> >& \_paraList){

if(now->token!="program\_head"){

cout << "getProgramHead error" << endl;

return;

}

获取主程序id及其行号;

调用getIdList获取主程序参数列表;//反转

}

#### 获取整个程序

\_Program\* getProgram(Type \*now){

if(now->token!="programstruct"){

cout << "getProgram error" << endl;

return NULL;

}

\_Program\* ASTRoot=new \_Program;

调用getProgramHead获取主程序头;

调用getProgramBody获取主程序体;

return ASTRoot;

}

## 实现细节

### 各节点类的行号设计

|  |  |
| --- | --- |
| 节点类 | 行号 |
| \_Expression：变量引用 | \_VariantReference所在行号 |
| \_Expression：无符号整数 | 无符号整数所在行号 |
| \_Expression：无符号浮点数 | 无符号浮点数所在行号 |
| \_Expression：字符常量 | 字符常量所在行号 |
| \_Expression：函数调用 | \_FunctionCall所在行号 |
| \_Expression：包含运算符的复合表达式 | 第一个操作数表达式的行号（递归定义） |
| \_FunctionCall | 函数名标识符所在行号 |
| \_VariantReference | 变量/常量/数组/函数名标识符所在行号 |
| \_Statement：\_Compound | BEGIN关键字所在行号 |
| \_Statement：\_RepeatStatement | REPEAT关键字所在行号 |
| \_Statement：\_WhileStatement | WHILE关键字所在行号 |
| \_Statement：\_ForStatement | FOR关键字所在行号 |
| \_Statement：\_IfStatement | IF关键字所在行号 |
| \_Statement：\_AssignStatement | :=赋值号所在行号 |
| \_Statement：\_ProcedureCall | 过程名标识符所在行号 |
| \_Type | 基本类型关键字所在行号 |
| \_FormalParameter | 参数名标识符所在行号 |
| \_FunctionDefinition | 函数名标识符所在行号 |
| \_Variant | 变量名标识符所在行号 |
| \_Constant | 常量名标识符所在行号 |
| \_SubProgram | 无行号 |
| \_Program | 主程序名所在行号 |

### 左递归与列表反转

PASCAL-S的文法中，涉及到递归的，均为左递归，所以在先序遍历语法分析树时，先识别的语法成分实际上在源代码中是后出现的，所以在列表识别完毕时，需要将列表进行反转，以符合实际的顺序。

另外需要特别注意的是，idList在一些地方只单独出现一次，例如主程序参数列表；而在一些地方会出现多次，例如子程序参数列表。在前者的情况中，最后需要反转列表；在后者的情况中，则不需要反转列表，反转列表的操作由更高层的语法成分完成。

### 常量值保存

常量值保存下来，一方面可以方便后续计算常量表达式的值、常量定义的常数传播、数组常量下标的越界检查等，另一方面是为了代码生成。考虑到这两方面的需求，所以在\_Constant节点中，有具体类型的域保存其具体的取值，另外还用一个string变量统一存储其字符串形式。

## 对外接口

### 入口

* Type\* ParseTreeHead，即普通抽象语法树的根节点

### 出口

* \_Program\* ASTRoot = getProgram(ParseTreeHead)，返回值保存在ASTRoot中，即抽象语法树的根节点