北京邮电大学

《编译原理与技术课程设计》  
报 告

**指导教师 ： 李文生**

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | **班级** | **学号** | **备注** |
|  |  |  | **组长** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**计算机学院**

**2017 年 4 月**

目录

[1.课程设计题目 4](#_Toc480974192)

[2.课程设计目标和要求 4](#_Toc480974193)

[2.1 文法 4](#_Toc480974194)

[3.需求分析 8](#_Toc480974195)

[3.1 数据流图 8](#_Toc480974196)

[3.2 功能说明 8](#_Toc480974197)

[3.3 数据说明 9](#_Toc480974198)

[4． 开发环境 9](#_Toc480974199)

[5． 总体设计 9](#_Toc480974200)

[5.1 数据结构设计 9](#_Toc480974201)

[5.2 总体结构设计 11](#_Toc480974202)

[5.2.1功能模块的划分 11](#_Toc480974203)

[5.2.2模块功能及模块间的关系 11](#_Toc480974204)

[5.2.3模块之间的接口 11](#_Toc480974205)

[5.3 用户接口设计 12](#_Toc480974206)

[6.各部分详细设计说明 12](#_Toc480974207)

[6.1 词法分析模块 13](#_Toc480974208)

[6.1.1 接口描述 13](#_Toc480974209)

[6.1.2 功能描述 13](#_Toc480974210)

[6.1.3 所用数据结构说明 13](#_Toc480974211)

[6.1.4 算法描述 14](#_Toc480974212)

[6.2 语法分析模块 14](#_Toc480974213)

[6.2.1 接口描述 14](#_Toc480974214)

[6.2.2 功能描述 14](#_Toc480974215)

[6.2.3 所用数据结构说明 14](#_Toc480974216)

[6.2.4 算法描述 16](#_Toc480974217)

[6.3 语义分析 16](#_Toc480974218)

[6.3.1 接口描述 16](#_Toc480974219)

[6.3.2 功能描述 17](#_Toc480974220)

[6.3.3 所用数据结构说明 17](#_Toc480974221)

[6.3.4 算法描述 19](#_Toc480974222)

[6.4 代码生成 20](#_Toc480974223)

[6.4.1 接口描述 20](#_Toc480974224)

[6.4.2 功能描述 20](#_Toc480974225)

[6.4.3 所用数据结构说明 21](#_Toc480974226)

[6.4.4 算法描述 21](#_Toc480974227)

[7.源程序清单 22](#_Toc480974228)

[8.测试报告 106](#_Toc480974229)

[8.1测试环境 106](#_Toc480974230)

[8.2测试计划及结果 106](#_Toc480974231)

[一、程序框架 107](#_Toc480974232)

[二、常量声明部分 111](#_Toc480974233)

[三、变量声明部分 116](#_Toc480974234)

[四、过程和函数声明部分 122](#_Toc480974235)

[五、函数和过程调用部分 128](#_Toc480974236)

[六、程序体部分 131](#_Toc480974237)

[9.实验总结 143](#_Toc480974238)

[9.1主要问题 143](#_Toc480974239)

[9.1.1 Flex和Yacc的连接问题 143](#_Toc480974240)

[9.1.2文法修改 143](#_Toc480974241)

[9.1.3文法扩充 144](#_Toc480974242)

[9.1.4错误处理 144](#_Toc480974243)

[9.1.5符号表操作 144](#_Toc480974244)

[9.2改进建议 144](#_Toc480974245)

[9.2.1符号表设计 144](#_Toc480974246)

[9.2.2记录类型的嵌套 144](#_Toc480974247)

[9.2.3词法分析错误处理 144](#_Toc480974248)

[9.2.4动态检查 145](#_Toc480974249)

[9.2.5 Yacc错误处理机制 145](#_Toc480974250)

[9.3心得体会 145](#_Toc480974251)

# 1.课程设计题目

Pascal-S语言编译程序的设计与实现

# 2.课程设计目标和要求

目标： 按照所给Pascal-S语言的语法，参考Pascal语言的语义，设计并实现Pascal-S语言的编译程序。

要求： a. 需求分析报告

b. 总体设计报告（软件功能描述、功能模块划分、软件结 构图、符号表结构设计、模块间接口定义等）

c. 详细设计报告（模块功能、输入/输出、处理逻辑等）

d. 编码实现（源程序、可执行程序）

## 2.1 文法

program → program\_head program\_body .

| program\_head program\_body

program\_head → program id ( identifier\_list ) ;

| program id ( identifier\_list ;

| program id identifier\_list ) ;

| program id ( ) ;

| program id ;

program\_body → const\_declarations

type\_declarations

var\_declarations

subprogram\_declarations

compound\_statement

| ε

identifier\_list → identifier\_list , id

| id

| identifier\_list error id

const\_declarations → const const\_declaration ;

| const const\_declaration

| ε

const\_declaration → const\_declaration ; id = const\_variable

| const\_declaration ; id := const\_variable

| id = const\_variable

| id := const\_variable

const\_variable → + id | - id | id

| + float | - float | float

| + int | - int | int

| letter

type\_declarations → type type\_declaration ;

| type type\_declaration

| ε

type\_declaration → type\_declaration ; id = type

| id = type

R → record

type → standrad\_type

| R record\_body end

| array [ periods ] of standard\_type

| id

standard\_type → integer | real | boolean | char

record\_body→ var\_declaretion | ε

periods → periods , period

| period

period → const\_variable . . const\_variable

| const\_variable error const\_variable

var\_declarations → var var\_declaration ;

| var var\_declaration

| ε

var\_declaration → var\_declaration ; identifier\_list : type

| var\_declaration ; identifier\_list

| identifier\_list

| identifier\_list : type

subprogram\_declarations → subprogram\_declarations subprogram\_declaration ;

| ε

subprogram\_declaration → subprogram\_head

const\_declarations

type\_declarations

var\_declarations

compound\_statement

subprogram\_head → M formal\_parameter : type ;

| M : type ;

| M formal\_parameter ;

| N formal\_parameter ;

| N ;

| N formal\_parameter : type ;

M → function id

N → procedure id

formal\_parameter → ( parameter\_lists )

| ( parameter\_lists ; )

| ε

parameter\_lists → parameter\_lists ; parameter\_list

| parameter\_lists error parameter\_list

| parameter\_list

parameter\_list → var\_parameter

| value\_parameter

var\_parameter → var value\_parameter

value\_parameter → identifier\_list : type

| identifier\_list

compound\_statement → begin statement\_list end

| begin statement\_list ; end

statement\_list → statement\_list ; statement

| statement\_list statement

| statement

statement → variable assignop expression

| variable relop expression

| variable error assignop expression

| call\_procedure\_statement

| compound\_statement

| if expression then statement else\_part

| S case\_body end

| S case\_body

| while expression do statement

| while expression statement

| repeat statement\_list until expression

| T assignop expression updown expression do statement

| T assignop expression updown expression statement

| ε

| write

| read

S → case expression of

write → write ( expression\_list )

| writeln ( expression\_list )

| writeln

read → read ( variable\_list )

T → for id

variable → IDM id\_varparts

IDM → id

id\_varparts → id\_varparts id\_varpart

| ε

id\_varpart → [ expression\_list ]

| . id

variable\_list → variable\_list , variable

| variable

else\_part → else statement

| ε

case\_body → branch\_list

| ε

branch\_list → branch\_list ; branch

| branch

branch → const\_list : statement

const\_list → const\_list , const\_variable

| const\_variable

updown → to | downto

call\_procedure\_statement → id

| id ( expression\_list )

| id expression\_list )

| id ( expression\_list

expression\_list → expression\_list , expression

| expression

expression → simple\_expression relop simple\_expression

| simple\_expression

simple\_expression → term | + term | - term

| simple\_expression addop term

| simple\_expression minusop term

term → term mulop factor

| factor

factor → unsign\_const\_variable

| variable

| id ( expression\_list )

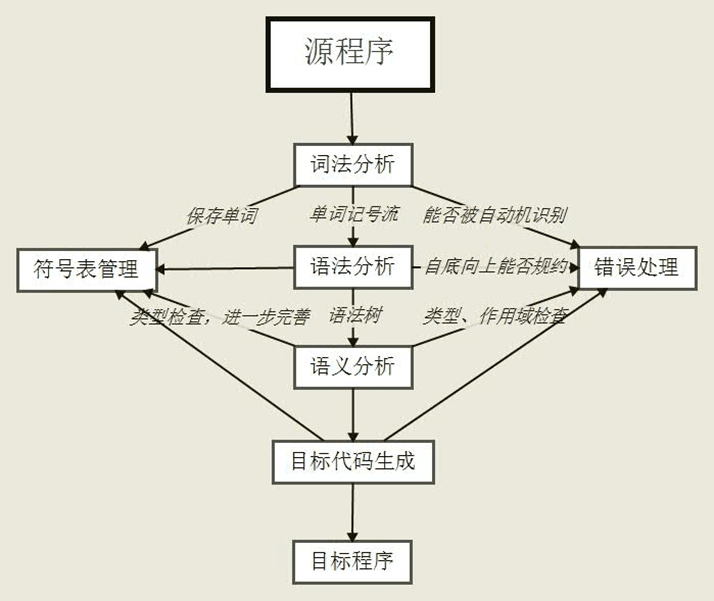
| ( expression )

| not factor

unsign\_const\_variable → float | int | letter

# 3.需求分析

## 3.1 数据流图



3.2 功能说明  
3.2.1词法分析  
1.分析PASCAL语言语句，通过与语法分析模块的接口将分析结果以记号流的形式交给Yacc语法分析程序处理分析。  
2.检查词法分析中的错误并报告。  
3.获取每个记号的value、row\_no、col\_no等属性，将这些属性传递给Yacc程序以便后面的处理。  
3.2.2 语法分析  
1.按照给出的语法产生式对记号进行移进规约操作，从而传递各语法记号的相关属性。  
2.建立函数表以及函数中的符号表。建立const和type的存储表。  
3.进行语法检查判断语法记号是否符合文法规定，进行错误处理恢复。  
3.2.3语义分析  
1.检查函数名、变量名、参数名、const名和type名是否重定义。  
2.检查函数名、变量名、参数名、const名和type名是否定义过。  
3.进行表达式的类型检查。  
4.检查参数的类型和参数数目的匹配。  
5.检查数组越界和数组维度错误问题。  
6.检查除法中除数为0的情况。

## 3.3 数据说明

词法分析调用yylex() 将单词记号流传递给语法分析，语法分析使用yylval将各语法记号的相关属性传递给语义分析进行各类检查，最后使用各语法记号的属性值进行目标代码生成。

# 4.开发环境

Linux Ubuntu16.04开发环境

Lex + Bison

# 5.总体设计

## 5.1 数据结构设计

//存储识别到数组的上下界  
struct array\_elem  
{  
int low\_bound;  
int up\_bound;  
};  
  
//函数表中符号表的结构  
struct symbol  
{  
string name; //符号名称  
string var\_type; //变量类型  
string array\_type;  
//string ret\_type; //函数返回值类型  
struct symbol\* parameters; //参数链表  
struct symbol\* members; //成员链表（记录类型）   
int dimension; //维数   
int line\_num; //引用行   
vector<struct array\_elem> bounds;  
int isVariable;//0：自定义type；1：变量id  
struct symbol\* next\_symbol;// 下一个符号   
};  
  
//函数表结构  
struct function  
{  
 string name; //函数名  
 string ret\_type; //返回值类型  
 int const\_start; //函数const常量开始下标  
 int type\_start; //函数type开始下标  
 int type\_num; //函数中定义的type的数量  
 int const\_num; //函数中定义的const的数量  
 struct symbol\* symbol\_list; //函数内部符号表   
 struct function\* next\_function; //指向下一个函数结点  
};  
  
//const常量存储结构  
struct const\_node  
{  
 string name; //const常量名字  
 string const\_type; //常量类型  
 string value; //常量的值  
};  
  
//type定义存储结构  
struct type\_node  
{  
 string name; //type定义的名字  
 string type; //type重定义的类型  
};  
  
记号的数据结构  
class token  
{  
 public:  
 int row\_no; //记号的行号  
 int col\_no; //记号的列号  
 string type; //记号的类型  
 string value; //记号的值  
 bool is\_var; //是否为var型变量  
 //构造函数  
 token()  
 {  
 type = "";  
 value = "";  
 is\_var = false;  
 }  
};

struct function\* func\_head\_node = NULL; //函数表头节点

struct function\* func\_cur\_node = NULL; //当前最新函数表节点

struct symbol\* sym\_head\_node = NULL; //变量头节点

struct symbol\* sym\_cur\_node = NULL; //当前最新变量节点

struct symbol\* param\_cur\_node = NULL; //当前最新参数节点

vector<const\_node> const\_table; //常量表

vector<type\_node> type\_table; //自定义类型表

## 5.2 总体结构设计

### 5.2.1功能模块的划分

a. 词法分析模块

b. 语法、语义分析模块

c. 目标代码生成模块

### 5.2.2模块功能及模块间的关系

a. 词法分析模块

利用lex工具对源代码进行词法分析处理，处理结果以<记号，属性>的形式传递给语法、语义分析模块，同时进行词法分析错误处理

b. 语法、语义分析模块

建立符号表，通过词法分析模块传递的记号流，根据LALR(1)文法和翻译方案，对符号栈和状态栈进行移进归约操作，向符号表中写入相应记号及属性值，同时进行语法分析错误处理，

类型检查、作用域检查和其他语义检查，进行其他错误处理。完善符号表。

c. 目标代码生成模块

根据语法分析和语义分析的结果，进行翻译动作，将Pascal-S语言翻译成C语言。

### 5.2.3模块之间的接口

a. 词法分析模块

void yylex()语法分析模块调用yylex()得到当前的记号及其属性值

b. 语法、语义分析模块

void yylex()语法分析模块调用yylex()得到当前的记号及其属性值，变量YYSTYPE yylval词法分析模块将记号的属性值存储在yylval中，语法分析程序使用yylval进行属性计算

## 5.3 用户接口设计

编辑一个名为Makefile的文件，将命令写到该文件中，在测试文件当前目录下打开终端，输入make，即可开始编译Lex和Yacc文件，生成可执行文件compiler。用户在file.txt中输入需要编译的PASCAL语言，在终端中输入命令./compiler即可以开始对源语言进行编译，输出的C语言翻译代码在output.txt中，错误信息会打印在屏幕上。

**Makefile文件**

LEX=flex

YACC=bison

CC=g++

OBJECT=compiler #生成的目标文件

$(OBJECT): lex.yy.o my\_yacc.tab.o

$(CC) lex.yy.o my\_yacc.tab.o -o $(OBJECT)

@./$(OBJECT) #编译后立刻运行

lex.yy.o: lex.yy.c my\_yacc.tab.h variable1.h

$(CC) -c lex.yy.c -ll -lfl

my\_yacc.tab.o: my\_yacc.tab.c

$(CC) -c my\_yacc.tab.c

my\_yacc.tab.c my\_yacc.tab.h: my\_yacc.y

# bison使用-d参数编译.y文件

$(YACC) -d my\_yacc.y

lex.yy.c: my\_lex.l

$(LEX) my\_lex.l

clean:

@rm -f $(OBJECT) \*.o

# 6.各部分详细设计说明

## 6.1 词法分析模块

### 6.1.1 接口描述

变量yytext

存储当前匹配的字符串

void lex\_return()

将识别到的记号值和各个属性传给yylval

int yylex()

由Yacc语法分析程序调用，取得记号进行语法分析

### 6.1.2 功能描述

对PASCAL程序进行处理，根据正则式匹配，将PASCAL代码转化为记号流形式传递给Yacc语法分析程序，Yacc程序使用yylex()函数来取得记号。

### 6.1.3 所用数据结构说明

//记号的数据结构

class token

{

public:

int row\_no; //记号的行号

int col\_no; //记号的列号

string type; //记号的类型

string value; //记号的值

bool is\_var; //是否为var型变量

//构造函数

token()

{

type = "";

value = "";

is\_var = false;

}

};

### 6.1.4 算法描述

编写lex文件，通过正则式进行识别原PASCAL语句，返回相应的token给Yacc语法分析程序。并且通过yylval将记号的各个属性也传给语法分析程序以便于后面的处理。语法分析程序使用yylex()调用词法分析过程，词法分析过程中发现了错误的记号则报错。

## 6.2 语法分析模块

### 6.2.1 接口描述

void yyparse();

主函数调用yyparse()函数对传递上来的记号流进行语法分析。

### 6.2.2 功能描述

对词法分析传来的记号流根据规定的文法进行语法分析，对不符合规定文法的情况进行报错，建立函数表、符号表、const表和type表，在识别到标志符语法记号时将其信息添加到各种符号表中。在规约过程中计算记号的属性并不断向上传递，以便后面的语义分析和代码生成。

### 6.2.3 所用数据结构说明

//存储识别到数组的上下界

struct array\_elem

{

int low\_bound;

int up\_bound;

};

//函数表中符号表的结构

struct symbol

{

string name; //符号名称

string var\_type; //变量类型

string array\_type;

//string ret\_type; //函数返回值类型

struct symbol\* parameters; //参数链表

struct symbol\* members; //成员链表（记录类型）

int dimension; //维数

int line\_num; //引用行

vector<struct array\_elem> bounds;

int isVariable;//0：自定义type；1：变量id

struct symbol\* next\_symbol;// 下一个符号

};

//函数表结构

struct function

{

string name; //函数名

string ret\_type; //返回值类型

int const\_start; //函数const常量开始下标

int type\_start; //函数type开始下标

int type\_num; //函数中定义的type的数量

int const\_num; //函数中定义的const的数量

struct symbol\* symbol\_list; //函数内部符号表

struct function\* next\_function; //指向下一个函数结点

};

//const常量存储结构

struct const\_node

{

string name; //const常量名字

string const\_type; //常量类型

string value; //常量的值

};

//type定义存储结构

struct type\_node

{

string name; //type定义的名字

string type; //type重定义的类型

};

//记号的数据结构

class token

{

public:

int row\_no; //记号的行号

int col\_no; //记号的列号

string type; //记号的类型

string value; //记号的值

bool is\_var; //是否为var型变量

//构造函数

token()

{

type = "";

value = "";

is\_var = false;

}

};

### 6.2.4 算法描述

编写Yacc文件，规定文法产生式，调用yylex()函数对从词法分析程序获取的记号进行移进-规约操作，在遇到标识符时进行添加符号表操作，建立起函数表、符号表、const表和type表。在规约时对产生式左边符号的属性进行计算，为后面的语义分析和代码生成提供基础。主函数调用yyparse()开始进行语法分析过程。

## 6.3 语义分析

### 6.3.1 接口描述

vector<string> split(string &str,string pattern);

//切分函数

void add\_parameter(string parameters);

//添加函数参数节点函数

void add\_symbol(string symbols, string type);

//添加符号节点函数（变量）

void add\_func(string name, string type, struct function \*currentNode);

//添加函数节点函数

void add\_const(string id, string replace, string replace\_type);

//添加常量函数

void add\_typedef(string id, string type);

//添加自定义类型

void redefine\_type(string type);

//函数返回类型写回函数

void check\_type\_error\_output(string type1, string type2, int rownum1, int colnum1, int rownum2, int colnum2);

//错误信息输出函数

int assign\_type\_check( string left\_type, string right\_type );

//赋值语句类型检查

int check\_in\_ids(string &id\_list);

//记号串中重定义检查

int check\_repeat\_definition(string &id\_list);

//全局重定义检查

int check\_var\_declaration(string &id\_list,string &type);

int check\_repeat\_func(string id);

//函数名重定义检查

int param\_type\_check ( string id, string params );

//函数调用参数检查

string del\_space(string str);

//去空格函数

string find\_func\_type ( string id );

//获取函数返回类型

string combine\_type( string type1, string type2 );

//表达式类型合并函数

string find\_local\_var\_func\_type( string id, string id\_varparts );

//局部变量或函数类型查找函数

string find\_global\_var\_type( string id, string id\_varparts );

//全局变量类型查找函数

string get\_const\_type(string id);

//常量类型获取函数

string get\_const\_value(string id);

//常量名字获取函数

string get\_typedef\_type(string id);

//自定义类型获取函数

string get\_typedef\_value(string id);

//自定义类型名获取函数

### 6.3.2 功能描述

检查函数名、变量名、参数名、const名和type名是否重定义；检查函数名、变量名、参数名、const名和type名是否定义过；进行表达式的类型检查；检查参数的类型和参数数目的匹配；检查数组越界和数组维度错误问题；检查除法中除数为0的情况。

进行相应的错误检查和处理。

### 6.3.3 所用数据结构说明

//存储识别到数组的上下界

struct array\_elem

{

int low\_bound;

int up\_bound;

};

//函数表中符号表的结构

struct symbol

{

string name; //符号名称

string var\_type; //变量类型

string array\_type;

//string ret\_type; //函数返回值类型

struct symbol\* parameters; //参数链表

struct symbol\* members; //成员链表（记录类型）

int dimension; //维数

int line\_num; //引用行

vector<struct array\_elem> bounds;

int isVariable;//0：自定义type；1：变量id

struct symbol\* next\_symbol;// 下一个符号

};

//函数表结构

struct function

{

string name; //函数名

string ret\_type; //返回值类型

int const\_start; //函数const常量开始下标

int type\_start; //函数type开始下标

int type\_num; //函数中定义的type的数量

int const\_num; //函数中定义的const的数量

struct symbol\* symbol\_list; //函数内部符号表

struct function\* next\_function; //指向下一个函数结点

};

//const常量存储结构

struct const\_node

{

string name; //const常量名字

string const\_type; //常量类型

string value; //常量的值

};

//type定义存储结构

struct type\_node

{

string name; //type定义的名字

string type; //type重定义的类型

};

//记号的数据结构

class token

{

public:

int row\_no; //记号的行号

int col\_no; //记号的列号

string type; //记号的类型

string value; //记号的值

bool is\_var; //是否为var型变量

//构造函数

token()

{

type = "";

value = "";

is\_var = false;

}

};

### 6.3.4 算法描述

在Yacc文件中利用语法制导翻译技术，对语法符号进行语义分析。

在函数头声明语句或者程序声明语句时，进行定位操作，识别到程序名标识符或者函数头函数名标识符时，先使用check\_repeat\_func 函数进行查重处理，然后使用add\_func函数添加函数表结点，将函数名和函数返回值写到函数表中，若重复则会报错。

当在参数列表语句规约时，同样进行定位操作，首先调用check\_in\_ids函数和check\_repeat\_definition函数对id\_list中的变量名进行查重操作，然后调用add\_parameter函数将参数信息存储到当前函数表第一个symbol结点的parameters链表中。若在查重处理时发现重复定义名字，则删去该名字，不予加入参数表。

当在var定义变量语句规约时，同样进行定位操作，首先调用check\_var\_declaration函数对id\_list中的变量名进行查重操作，在此函数中的add\_symbol函数将变量信息存储到当前函数表结点的symbol\_list链表中。若在查重处理时发现重复定义名字，则删去该名字，不予加入符号表，并报错。

在const定义常量语句规约时，进行定位操作，首先调用check\_repeat\_definition函数对ID进行查重处理，若不重复才调用add\_const函数将常量定义的各信息添加入const\_table表中，重复定义则不加入const\_table表中并报错。在识别到一个函数名时，该函数const\_start值为当前const\_table表最后一个下标+1,根据每个函数的const\_num值可以对const\_table进行遍历。

在type定义常量语句规约时，进行定位操作，首先调用check\_repeat\_definition函数对ID进行查重处理，若不重复才调用add\_typedef函数将常量定义的各信息添加入type\_table表中，重复定义则不加入type\_table表中并报错。在识别到一个函数名时，该函数type\_start值为当前type\_table表最后一个下标+1,根据每个函数的type\_num值可以对type\_table进行遍历。

在规约变量名使用时，通过find\_local\_var\_func\_type函数对当前函数结点中的const\_table和symbol符号表进行重定位操作，查找变量名是否定义，find\_local\_var\_func\_type函数在当前函数结点找到则直接返回找到的变量类型，找不到则返回函数find\_global\_var\_type的返回值，即在主函数结点中查找变量名。

在规约函数名使用时，进行重定位操作，通过函数find\_func\_type在函数表中寻找调用函数名是否定义，找到则返回其类型，找不到则返回NULL。

在规约expression\_list参数调用时，使用函数param\_type\_check对参数的类型是否匹配和参数的数量是否相同进行检查。若不匹配或者数量不同则报错。

在规约赋值语句时，使用assign\_type\_check函数对两边类型的匹配进行检查。

在规约乘除法或者加减法语句时，表达式的类型通过调用combine\_type函数确定，要根据运算符两边的操作数决定。

在规约while和if语句时，判断条件表达式是否为bool类型，如果不为bool类型则进行报错。

## 6.4 代码生成

### 6.4.1 接口描述

string definition\_trans(string para, bool is\_para);

//对参数中和var变量定义语句进行翻译

string write\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);

//对write语句进行翻译

string writeln\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);

//对writeln语句进行翻译

string read\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);

//对read语句进行翻译

### 6.4.2 功能描述

根据语法分析得到的记号属性和语义分析进行检查的结果，根据相应的翻译动作，得到目标语言代码。

### 6.4.3 所用数据结构说明

//记号的数据结构

class token

{

public:

int row\_no; //记号的行号

int col\_no; //记号的列号

string type; //记号的类型

string value; //记号的值

bool is\_var; //是否为var型变量

//构造函数

token()

{

type = "";

value = "";

is\_var = false;

}

};

### 6.4.4 算法描述

string definition\_trans(string para, bool is\_para);

函数对参数或者var定义变量语句进行翻译，输入为“type:id\_list”，返回C语言声明语句。一个传入的para字符串前若有"var+"，则说明为参数中传引用的参数，则去掉前面的标记置变量is\_var为true。若is\_para为true，则为参数的翻译，因为参数中不允许有id\_list，只允许出现单个id，则后面的操作和var定义翻译有所不同。将字符串按照冒号分开后，再将变量名以逗号分开处理。同时对类型中的array数组类型进行C语言的翻译。传引用定义需要在每个变量前都加上类型。

string write\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);

函数对write语句进行翻译，在规约write语句之前已经在expression\_list处进行了是否定义的检查，此处则不需要查询。将exp\_value用\*分开，并将exp\_type类型翻译成C语言格式控制，若exp\_type为NULL，则删除此value。最后将各部分拼接输出。

string writeln\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);

功能与write\_trans函数相同，在格式控制最后加上\n即可。

string read\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);

函数对read语句进行翻译，在规约read语句之前已经在expression\_list处进行了是否定义的检查，此处则不需要查询。将exp\_value用\*分开，在前面加上&，并将exp\_type类型翻译成C语言格式控制，若exp\_type为NULL，则删除此value。最后将各部分拼接输出。

# 7.源程序清单

**my\_lex.l**

%{

//需要包括的头文件已经自定义的函数和变量的声明

#include <stdio.h>

#include <stdlib.h>

#include<math.h>

#include "variable1.h"

#include "my\_yacc.tab.h"

int id\_num=0;

int num\_num=0;

int digit=0;

int row\_no = 1;

int col\_no = 1;

char \*curfilename;

void wrong();

void lex\_return();

string big2low(string bigstr);

extern "C"

{

int yywrap(void);

int yylex(void);

}

%}

/\*正则表达式识别换行符、标识符、数字和字符等\*/

delim [\t]

ws {delim}+

letter [A-Za-z]

digit [0-9]

digits {digit}+

id {letter}({letter}|{digit})\*

num {digits}(\.{digits})?(E[+\-]?{digits})?

newline [\n]

%%

/\*关键字的识别\*/

[Bb][Ee][Gg][Ii][Nn] {lex\_return(); return(BEGIN1);}

[pP][rR][oO][gG][rR][aA][mM] {lex\_return(); return(PROGRAM);}

[vV][aA][rR] {lex\_return(); return(VAR);}

[aA][rR][rR][aA][yY] {lex\_return(); return(ARRAY);}

[rR][eE][cC][oO][rR][dD] {lex\_return(); return(RECORD);}

[eE][nN][dD] {lex\_return(); return(END);}

[oO][fF] {lex\_return(); return(OF);}

[iI][nN][tT][eE][gG][eE][rR] {lex\_return(); return(INTEGER);}

[rR][eE][aA][lL] {lex\_return(); return(REAL);}

[bB][oO][oO][lL][eE][aA][nN] {lex\_return(); return(BOOLEAN);}

[fF][uU][nN][cC][tT][iI][oO][nN] {lex\_return(); return(FUNCTION);}

[pP][rR][oO][cC][eE][dD][uU][rR][eE] {lex\_return(); return(PROCEDURE);}

[iI][fF] {lex\_return(); return(IF);}

[tT][hH][eE][nN] {lex\_return(); return(THEN);}

[eE][lL][sS][eE] {lex\_return(); return(ELSE);}

[wW][hH][iI][lL][eE] {lex\_return(); return(WHILE);}

[dD][oO] {lex\_return(); return(DO);}

[rR][eE][aA][dD] {lex\_return(); return(READ);}

[wW][rR][iI][tT][eE] {lex\_return(); return(WRITE);}

[nN][oO][tT] {lex\_return(); return(NOT);}

[tT][rR][uU][eE] {lex\_return(); return(TRUE);}

[fF][aA][lL][sS][eE] {lex\_return(); return(FALSE);}

[fF][oO][rR] {lex\_return(); return(FOR);}

[tT][oO] {lex\_return(); return(TO);}

[cC][oO][nN][Ss][Tt] {lex\_return(); return(CONST);}

[tT][Yy][Pp][Ee] {lex\_return(); return(TYPE);}

[cC][hH][Aa][Rr] {lex\_return(); return(CHAR);}

[Rr][Ee][Pp][Ee][Aa][Tt] {lex\_return(); return(REPEAT);}

[Uu][Nn][Tt][Ii][Ll] {lex\_return(); return(UNTIL);}

[Dd][Oo][Ww][Nn][Tt][Oo] {lex\_return(); return(DOWNTO);}

[Cc][Aa][Ss][Ee] {lex\_return(); return(CASE);}

[Ww][Rr][Ii][Tt][Ee][Ll][Nn] {lex\_return(); return(WRITELN);}

"=" {lex\_return(); return(RELOP);}

"<>" {lex\_return(); return(RELOP);}

"<" {lex\_return(); return(RELOP);}

">" {lex\_return(); return(RELOP);}

">=" {lex\_return(); return(RELOP);}

"<=" {lex\_return(); return(RELOP);}

"+" {lex\_return(); return(ADDOP);}

"-" {lex\_return(); return(MINUSOP);}

"\*" {lex\_return(); return(MULOP);}

"/" {lex\_return(); return(MULOP);}

":=" {lex\_return(); return(ASSIGNOP);}

"," {lex\_return(); return(COMMA);}

";" {lex\_return(); return(SEMICOLON);}

":" {lex\_return(); return(COLON);}

"[" {lex\_return(); return(LEFTCUBE);}

"]" {lex\_return(); return(RIGHTCUBE);}

"(" {lex\_return(); return(LEFTCIRCLE);}

")" {lex\_return(); return(RIGHTCIRCLE);}

"." {lex\_return(); return(PROGRAMEND);}

or {lex\_return(); return(ADDOP);}

div {lex\_return(); return(MULOP);}

mod {lex\_return(); return(MULOP);}

and {lex\_return(); return(MULOP);}

[/']{letter}[/'] {lex\_return(); return(LETTER);}

{id} {id\_num++; lex\_return(); return (ID);}

{digits} {digit++; lex\_return(); return (INT);}

{num} {num\_num++; lex\_return(); return (FLOAT);}

{ws} {col\_no=col\_no+yyleng; }

{newline} {row\_no++; col\_no=1;}

/\*单行注释操作\*/

"{" {

char c;

c = yyinput();

col\_no++;

while(c != '}')

{

if(c == '\n') {row\_no++; col\_no=1; wrong();break;}/\*可能出现注释错误\*/

c = yyinput();

col\_no++;

}

}

/\*多行注释操作\*/

"(\*" {

printf("注释开始\n");

char c;

c = yyinput();

col\_no++;

printf("%c",c);

while(1)

{

if(c=='\*'){

c = yyinput();

printf("%c",c);

if(c==')'){c = yyinput();printf("%c",c);break;}

}

else {c = yyinput();

printf("%c",c);

if(c == '\n'){row\_no++; col\_no=1; }/\*可以接受多行注释\*/

}

}

printf("\n注释结束啦！\n");

row\_no++;

}

" " {col\_no=col\_no+yyleng;}

"\r" {col\_no=col\_no+yyleng;}

. {wrong();}

%%

/\*错误信息输出\*/

void wrong()

{

cout << "line " << row\_no << " column " << col\_no << " error message: find illegal syntax!" << endl;

return;

}

/\*通过yytext将识别到的字符串送给yylval，其结构在variable.h中定义，默认为int\*/

void lex\_return()

{

yylval.value = big2low(yytext);

yylval.row\_no = row\_no;

yylval.col\_no = col\_no;

col\_no=col\_no+yyleng;

}

/\*将字符串都转成小写字母\*/

string big2low(string bigstr)

{

string lowstr;

for ( int i = 0; i < bigstr.length(); i ++)

{

if ( bigstr[i] >= 'A' && bigstr[i] <= 'Z')

bigstr[i] = bigstr[i] + 32;

}

return bigstr;

}

int yywrap(void)

{

return 1;

}

**my\_yacc.y**

%{

#include "variable1.h"

const char\* sFile="file.txt"; //打开要读取的文本文件

ofstream output("output.txt");

int yydebug = 1;

extern "C"

{

extern int yylex(void);

void yyerror (char \*s);

}

vector<string> split(string &str,string pattern); //切分函数

void add\_parameter(string parameters); //添加函数参数节点函数

void add\_symbol(string symbols, string type); //添加符号节点函数（变量）

void add\_func(string name, string type, struct function \*currentNode); //添加函数节点函数

void add\_const(string id, string replace, string replace\_type); //添加常量函数

void add\_typedef(string id, string type); //添加自定义类型

void redefine\_type(string type); //函数返回类型写回函数

void check\_type\_error\_output(string type1, string type2, int rownum1, int colnum1, int rownum2, int colnum2);//错误信息输出函数

int assign\_type\_check( string left\_type, string right\_type ); //赋值语句类型检查

int check\_in\_ids(string &id\_list); //记号表中重定义检查

int check\_repeat\_definition(string &id\_list); //全局重定义检查

int check\_var\_declaration(string &id\_list,string &type);

int check\_repeat\_func(string id); //函数名重定义检查

int param\_type\_check ( string id, string params ); //函数调用参数检查

string del\_space(string str); //去空格函数

string find\_func\_type ( string id ); //获取函数返回类型

string combine\_type( string type1, string type2 ); //表达式类型合并函数

string find\_local\_var\_func\_type( string id, string id\_varparts ); //局部变量或函数类型查找函数

string find\_global\_var\_type( string id, string id\_varparts ); //全局变量类型查找函数

string definition\_trans(string para, bool is\_para); //声明语句翻译函数

string write\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no); //write语句翻译函数

string writeln\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no);//writeln语句翻译函数

string read\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no); //read语句翻译函数

string get\_const\_type(string id); //常量类型获取函数

string get\_const\_value(string id); //常量名字获取函数

string get\_typedef\_type(string id); //自定义类型获取函数

string get\_typedef\_value(string id); //自定义类型名获取函数

struct function\* func\_head\_node = NULL; //函数表头节点

struct function\* func\_cur\_node = NULL; //当前最新函数表节点

struct symbol\* sym\_head\_node = NULL; //变量头节点

struct symbol\* sym\_cur\_node = NULL; //当前最新变量节点

struct symbol\* param\_cur\_node = NULL; //当前最新参数节点

struct function\* call\_procedure\_id\_check( string id ); //函数调用时对函数名的检查

string int2str( int int\_temp ); //整型转字符串函数

string cur\_case\_type = ""; //case类型

vector<string> errors; //错误信息存储缓冲区

vector<string> nums; //case分支的常量值记录区

vector<string> branches; //case分支的语句记录区

vector<const\_node> const\_table; //常量表

vector<type\_node> type\_table; //自定义类型表

string array\_id = "";

int in\_record = 0; //是否在一个记录内部

int recover = 0; //恢复变量类型

int const\_cur\_pos = 0; //当前最新常量在常量表中的位置

int type\_cur\_pos = 0; //当前最新类型在自定义类型表中的位置

int dim = 0;

int is\_main = 1;

int time\_r = 1;

%}

%token PROGRAM 1

%token FUNCTION 2

%token PROCEDURE 3

%token INTEGER 4

%token REAL 5

%token BOOLEAN 6

%token ARRAY 7

%token RECORD 8

%token VAR 9

%token BEGIN1 10

%token END 11

%token OF 12

%token IF 13

%token THEN 14

%token ELSE 15

%token WHILE 16

%token DO 17

%token NOT 18

%token TRUE 19

%token FALSE 20

%token READ 21

%token WRITE 22

%token COMMA 37

%token SEMICOLON 38

%token COLON 39

%token LEFTCUBE 40

%token RIGHTCUBE 41

%token LEFTCIRCLE 42

%token RIGHTCIRCLE 43

%token PROGRAMEND 45

%token FLOAT 46

%token INT 47

%token ID 48

%token RELOP 49

%token ADDOP 50

%token MULOP 51

%token ASSIGNOP 52

%token SYMBOL 53

%token FOR 54

%token TO 55

%token CONST 57

%token MINUSOP 58

%token TYPE 59

%token CHAR 60

%token REPEAT 61

%token UNTIL 62

%token DOWNTO 63

%token CASE 64

%token LETTER 65

%token WRITELN 66

%token READLN 67

%left RELOP

%left ADDOP

%left MULOP

%right SYMBOL

%nonassoc IFX

%nonassoc ELSE

%%

program : program\_head program\_body PROGRAMEND //完成对整个程序的分析和翻译

{

for (int i = 0; i < errors.size(); i ++) //显示程序中所有出现的错误

{

cout << errors[i] << endl;

}

cout << "\n" << "\n";

$$.value = "#include<stdio.h> \n" + $2.value;

output << $$.value << endl;

cout << $$.value << endl;

}

| program\_head program\_body //错误：程序结尾丢失.符号

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no+1) + " ";

string error = "error message: expected symbol of the end of the program!";

errors.push\_back(line + error);

cout << "\n" << "\n";

$$.value = "#include<stdio.h> \n" + $2.value;

output << $$.value << endl;

cout << $$.value << endl;

}

;

program\_head : PROGRAM ID LEFTCIRCLE identifier\_list RIGHTCIRCLE SEMICOLON //完成程序头的分析和翻译

{

$$.value = $2.value;

add\_func($2.value,"void",func\_cur\_node);

}

| PROGRAM ID LEFTCIRCLE identifier\_list SEMICOLON //错误：丢失）

{

string line = "line " + int2str($4.row\_no) + " column " + int2str($4.col\_no+1) + " ";

string error = "error message: missing ) !";

errors.push\_back(line + error);

$$.value = $2.value;

add\_func($2.value,"void",func\_cur\_node);

}

| PROGRAM ID identifier\_list RIGHTCIRCLE SEMICOLON //错误：丢失（

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no+1) + " ";

string error = "error message: missing ( !";

errors.push\_back(line + error);

$$.value = $2.value;

add\_func($2.value,"void",func\_cur\_node);

}

| PROGRAM ID LEFTCIRCLE RIGHTCIRCLE SEMICOLON //错误：丢失程序参数

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: expected content in the circles!";

errors.push\_back(line + error);

$$.value = $2.value;

add\_func($2.value,"void",func\_cur\_node);

}

| PROGRAM ID SEMICOLON //错误：丢失程序参数

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: expected content in the circles!";

errors.push\_back(line + error);

$$.value = $2.value;

add\_func($2.value,"void",func\_cur\_node);

}

;

program\_body : const\_declarations type\_declarations var\_declarations subprogram\_declarations compound\_statement //完成对程序体的分析和翻译

{

if (is\_main == 1)

{

$$.value = $1.value + $2.value + $3.value + $4.value + "\nint main()\n{\n\t" + $5.value + "\n\treturn 0;\n}";

}

else

{

$$.value = "\n{\n\t" + $1.value + $2.value + $3.value + $4.value + $5.value + "\n}";

}

}

;

identifier\_list : identifier\_list COMMA ID //记号列表，在定义参数等地方用到

{

$$.value = $1.value + ", " + $3.value;

}

| identifier\_list error ID //错误：记号之间的分隔符不恰当

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no+1) + " ";

string error = "error message: comma error!";

errors.push\_back(line + error);

$$.value = $1.value + ", " + $3.value;

}

| ID

{

$$.value = $1.value;

}

;

const\_declarations : CONST const\_declaration SEMICOLON //常量声明

{

if ($2.value == "")

{

$$.value = "\n";

}

else

{

$$.value = $2.value + ";\n";

}

}

| CONST const\_declaration //错误：缺少常量声明最后的分号

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no+1) + " ";

string error = "error message: expected semicolon at the end of the declaration!";

errors.push\_back(line + error);

if ($2.value == "")

{

$$.value = "\n";

}

else

{

$$.value = $2.value + ";\n";

}

}

| //无常量声明

{

$$.value = "";

}

;

const\_declaration : const\_declaration SEMICOLON ID RELOP const\_variable //常量声明，const\_variable为常量值

{

if ($5.value == "not exist") //错误：常量值未找到或未定义

{

string line = "line " + int2str($5.row\_no) + " column " + int2str($5.col\_no) + " ";

string error = "error message: const\_variable has not been defined!";

errors.push\_back(line + error);

$$.value = $1.value;

}

else

{

if (check\_repeat\_definition($3.value) == 0) //重定义检查：对已定义的函数名、参数名、const、type和变量名进行查重，有重复则删除id，返回值为0表示有重复，返回值为1表示无重复

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

else

{

add\_const($3.value,$5.value,$5.type); //查重后加入常量表

}

if ($3.value != "")

{

$$.value = $1.value + ";\nconst " + $5.type + " " + $3.value + " = " + $5.value;

}

else

{

$$.value = $1.value;

}

}

}

| const\_declaration SEMICOLON ID ASSIGNOP const\_variable //错误：等号误用成赋值号

{

string line = "line " + int2str($4.row\_no) + " column " + int2str($4.col\_no) + " ";

string error = "error message: using assignop when defining const!";

errors.push\_back(line + error);

if ($5.value == "not exist") //错误：常量值未找到或未定义

{

string line = "line " + int2str($5.row\_no) + " column " + int2str($5.col\_no) + " ";

string error = "error message: const\_variable has not been defined!";

errors.push\_back(line + error); //保存错误信息

$$.value = $1.value;

}

else

{

if (check\_repeat\_definition($3.value) == 0) //重定义检查：对已定义的函数名、参数名、const、type和变量名进行查重，有重复则删除id，返回值为0表示有重复，返回值为1表示无重复

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

else

{

add\_const($3.value,$5.value,$5.type); //查重后加入常量表

}

if ($3.value != "")

{

$$.value = $1.value + ";\nconst " + $5.type + " " + $3.value + " = " + $5.value;

}

else

{

$$.value = $1.value;

}

}

}

| ID RELOP const\_variable //常量声明

{

func\_cur\_node->const\_start = const\_cur\_pos;

if ($3.value == "not exist") //错误：常量值未找到或未定义

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: const\_variable has not been defined!";

errors.push\_back(line + error);

$$.value = "";

}

else

{

if (check\_repeat\_definition($1.value) == 0) //重定义检查：对已定义的函数名、参数名、const、type和变量名进行查重，有重复则删除id，返回值为0表示有重复，返回值为1表示无重复

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

else

{

add\_const($1.value,$3.value,$3.type); //查重后加入常量表

}

if ($1.value != "")

{

$$.value = "const " + $3.type + " " + $1.value + " = " + $3.value;

}

else

{

$$.value = "";

}

}

}

| ID ASSIGNOP const\_variable //错误：等号误用成赋值号

{

func\_cur\_node->const\_start = const\_cur\_pos;

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: using assignop when defining const!";

errors.push\_back(line + error);

if ($3.value == "not exist")

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: const\_variable has not been defined!";

errors.push\_back(line + error);

$$.value = "";

}

else

{

if (check\_repeat\_definition($1.value) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

else

{

add\_const($1.value,$3.value,$3.type);

}

if ($1.value != "")

{

$$.value = "const " + $3.type + " " + $1.value + " = " + $3.value;

}

else

{

$$.value = "";

}

}

}

;

const\_variable : ADDOP ID //常量类型：+id型

{

$$.value = get\_const\_value($2.value);

$$.type = get\_const\_type($2.value);

}

| MINUSOP ID //常量类型：-id型

{

$$.value = "-" + get\_const\_value($2.value);

$$.type = get\_const\_type($2.value);

}

| ID //常量类型：id型

{

$$.value = get\_const\_value($1.value);

$$.type = get\_const\_type($1.value);

}

| ADDOP FLOAT //常量类型：+浮点型

{

$$.value = $2.value;

$$.type = "float";

}

| MINUSOP FLOAT //常量类型：-浮点型

{

$$.value = "-" + $2.value;

$$.type = "float";

}

| FLOAT //常量类型：浮点型

{

$$.value = $1.value;

$$.type = "float";

}

| ADDOP INT //常量类型：+整型

{

$$.value = $2.value;

$$.type = "int";

}

| MINUSOP INT //常量类型：-整型

{

$$.value = "-" + $2.value;

$$.type = "int";

}

| INT //常量类型：整型

{

$$.value = $1.value;

$$.type = "int";

}

| LETTER //常量类型：字符型

{

$$.value = $1.value;

$$.type = "char";

}

;

type\_declarations : TYPE type\_declaration SEMICOLON //类型声明

{

if ($2.value == "")

{

$$.value = "\n";

}

else

{

$$.value = $2.value + ";\n";

}

}

| TYPE type\_declaration //错误：丢失末位分号

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no+1) + " ";

string error = "error message: expected semicolon at the end of the declaration!";

errors.push\_back(line + error);

if ($2.value == "")

{

$$.value = "\n";

}

else

{

$$.value = $2.value + ";\n";

}

}

| //无类型声明

{

$$.value = "";

}

;

type\_declaration : type\_declaration SEMICOLON ID RELOP type //类型声明

{

if ($5.value == "not exist")

{

string line = "line " + int2str($5.row\_no) + " column " + int2str($5.col\_no) + " ";

string error = "error message: the type has not been defined!";

errors.push\_back(line + error);

$$.value = $1.value;

}

else

{

if (check\_repeat\_definition($3.value) == 0)

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

else

{

add\_typedef($3.value,$5.type); //查重后加入类型表

}

if ($3.value != "")

{

$$.value = $1.value + ";\ntypedef " + $5.value + " " + $3.value;

}

else

{

$$.value = $1.value;

}

}

}

| ID RELOP type //类型声明

{

func\_cur\_node->type\_start = type\_cur\_pos;

if ($3.value == "not exist")

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: the type has not been defined!";

errors.push\_back(line + error);

$$.value = "";

}

else

{

if (check\_repeat\_definition($1.value) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

else

{

add\_typedef($1.value,$3.value); //查重后加入类型表

}

if ($1.value != "")

{

$$.value = "typedef " + $3.value + " " + $1.value;

}

else

{

$$.value = "";

}

}

}

;

R : RECORD

{

in\_record = 1; //在var\_declaration处规约时识别在record内部

}

;

type : standard\_type //类型种类：标准类型

{

$$.value = $1.value;

$$.type = $1.type;

}

| R record\_body END //记录类型

{

$$.value = "struct{" + $2.value + "}";

$$.type = "struct";

in\_record = 0;

}

| ARRAY LEFTCUBE periods RIGHTCUBE OF standard\_type //数组类型

{

$$.value = "array" + $6.value + "[" + $3.value + "]";

$$.type = "array" + $6.type;

}

| ID //已经声明的自定义类型

{

$$.type = get\_typedef\_type($1.value);

$$.value = $$.type;

}

;

standard\_type : INTEGER //基本类型

{

$$.value = "int";

$$.type = "int";

}

| REAL //浮点类型

{

$$.value = "float";

$$.type = "float";

}

| BOOLEAN //布尔类型

{

$$.value = "bool";

$$.type = "bool";

}

| CHAR //字符型

{

$$.value = "char";

$$.type = "char";

}

;

record\_body : var\_declaration //记录类型的内容部分

{

$$.value = $1.value;

}

|

{

$$.value = "";

}

;

periods : periods COMMA period //数组类型多个维度

{

$$.value = $1.value + "," + $3.value;

}

| period

{

$$.value = $1.value;

}

;

period : const\_variable PROGRAMEND PROGRAMEND const\_variable //数组类型每个维度的上下界

{

if (atoi($1.value.c\_str()) > atoi($4.value.c\_str()))

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: low bound is greater than up bound!";

errors.push\_back(line + error);

}

$$.value = $1.value + "|" + $4.value;

}

| const\_variable error const\_variable //上下界之间的分隔符使用非法

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no-1) + " ";

string error = "error message: error in the array!";

errors.push\_back(line + error);

$$.value = $1.value + "|" + $3.value;

}

;

var\_declarations : VAR var\_declaration SEMICOLON //变量声明

{

if ($2.value == "")

{

$$.value = "\n";

}

else

{

$$.value = $2.value + ";\n";

}

}

| VAR var\_declaration //变量声明最后缺少分号

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no+1) + " ";

string error = "error message: expected semicolon at the end of the declaration!";

errors.push\_back(line + error);

if ($2.value == "")

{

$$.value = "\n";

}

else

{

$$.value = $2.value + ";\n";

}

}

| //无变量声明

{

if (is\_main == 1)

{

add\_parameter("");

func\_head\_node -> symbol\_list -> next\_symbol = NULL;

}

else

{

func\_cur\_node -> symbol\_list -> next\_symbol = NULL;

}

$$.value = "\n";

}

;

var\_declaration : var\_declaration SEMICOLON identifier\_list COLON type

{

int repeat = 1;

if (in\_record == 1)

{

if (check\_in\_ids($3.value) == 0)

{

repeat = 0;

}

string former;

int pos = $1.value.find(" ");

former = $1.value.substr(pos+1);

string new\_list;

vector<string> ids;

if ($3.value != "")

{

new\_list = del\_space($3.value);

}

else

{

new\_list = "";

}

ids = split(new\_list,",");

for (int i = 0; i < ids.size(); i ++)

{

vector<string>::iterator it;

if (former.find(ids[i]) != -1)

{

it = ids.begin() + i;

ids.erase(it);

repeat = 0;

}

}

$3.value = "";

for (int i = 0; i < ids.size(); i ++)

{

$3.value = $3.value + ids[i] + ", ";

}

$3.value = $3.value.substr(0,$3.value.length() - 2);

if (repeat == 0)

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

}

else

{

if (check\_in\_ids($3.value) == 0 || check\_repeat\_definition($3.value) == 0)

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

if ($3.value != "")

{

add\_symbol($3.value, $5.value);

}

}

if ($3.value == "")

{

if ($1.value == "")

{

$$.value = "";

}

else

{

$$.value = $1.value;

}

}

else

{

if ($1.value == "")

{

$$.value = definition\_trans($5.value + ":" + $3.value,false);

}

else

{

$$.value = $1.value + ";" + definition\_trans($5.value + ":" + $3.value,false);

}

}

}

| var\_declaration SEMICOLON identifier\_list

{

int repeat = 1;

if (in\_record == 1)

{

if (check\_in\_ids($3.value) == 0)

{

repeat = 0;

}

string former;

int pos = $1.value.find(" ");

former = $1.value.substr(pos+1);

string new\_list;

vector<string> ids;

if ($3.value != "")

{

new\_list = del\_space($3.value);

}

else

{

new\_list = "";

}

ids = split(new\_list,",");

for (int i = 0; i < ids.size(); i ++)

{

vector<string>::iterator it;

if (former.find(ids[i]) != -1)

{

it = ids.begin() + i;

ids.erase(it);

repeat = 0;

}

}

$3.value = "";

for (int i = 0; i < ids.size(); i ++)

{

$3.value = $3.value + ids[i] + ", ";

}

$3.value = $3.value.substr(0,$3.value.length() - 2);

if (repeat == 0)

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

}

else

{

if (check\_in\_ids($3.value) == 0 || check\_repeat\_definition($3.value) == 0)

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

if ($3.value != "")

{

add\_symbol($3.value, "int");

}

}

if ($3.value == "")

{

if ($1.value == "")

{

$$.value = "";

}

else

{

$$.value = $1.value;

}

}

else

{

if ($1.value == "")

{

$$.value = definition\_trans("int:" + $3.value,false);

}

else

{

$$.value = $1.value + ";" + definition\_trans("int:" + $3.value,false);

}

}

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: variable type missing!";

errors.push\_back(line + error);

}

| identifier\_list

{

string tmp = "int";

if (is\_main == 1)

{

if (func\_cur\_node == func\_head\_node && time\_r == 1)

{

add\_parameter("");

time\_r = 0;

}

if (check\_var\_declaration($1.value,tmp) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

}

else

{

if (check\_var\_declaration($1.value,tmp) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

}

if ($1.value == "")

{

$$.value = "";

}

else

{

$$.value = definition\_trans("int:" + $1.value,false);

}

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: variable type missing!";

errors.push\_back(line + error);

}

| identifier\_list COLON type//not conflict with the parameters of the function

{

if (is\_main == 1)

{

if ( func\_cur\_node == func\_head\_node && time\_r == 1)

{

add\_parameter("");

time\_r = 0;

}

if (check\_var\_declaration($1.value,$3.value) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

}

else

{

if (check\_var\_declaration($1.value,$3.value) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable!";

errors.push\_back(line + error);

}

}

if ($1.value == "")

{

$$.value = "";

}

else

{

$$.value = definition\_trans($3.value + ":" + $1.value,false);

}

}

;

subprogram\_declarations : subprogram\_declarations subprogram\_declaration SEMICOLON

{

$$.value = $1.value + "\n" + $2.value;

}

|

{

$$.value = "";

}

;

subprogram\_declaration : subprogram\_head const\_declarations type\_declarations var\_declarations compound\_statement

{

is\_main = 1;

$$.value = $1.value + "\n{\n\t" + $2.value + $3.value + $4.value + $5.value + "\n}\n";

}

;

subprogram\_head : M formal\_parameter COLON type SEMICOLON

{

redefine\_type($4.value);

$$.value = $4.value + " " + $1.value + $2.value;

}

| M COLON type SEMICOLON

{

add\_parameter("");

$$.value = $3.value + " " +$1.value;

}

| M formal\_parameter SEMICOLON

{

redefine\_type("int");

$$.value = "int " + $1.value + $2.value;

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: function must have a return type!";

errors.push\_back(line + error);

}

| N formal\_parameter SEMICOLON

{

redefine\_type("void");

$$.value = "void " + $1.value + $2.value;

}

| N SEMICOLON

{

add\_parameter("");

$$.value = "void " + $1.value;

}

| N formal\_parameter COLON type SEMICOLON

{

redefine\_type("void");

$$.value = "void " + $1.value + $2.value;

string line = "line " + int2str($4.row\_no) + " column " + int2str($4.col\_no) + " ";

string error = "error message: a procedure doesn't have a return type!";

errors.push\_back(line + error);

}

;

M : FUNCTION ID

{

if (check\_repeat\_func($2.value) == 0)

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: the name of the function has been defined before!";

errors.push\_back(line + error);

}

add\_func($2.value,"integer",func\_cur\_node);

$$.value = $2.value;

is\_main = 0;

}

;

N : PROCEDURE ID

{

if (check\_repeat\_func($2.value) == 0)

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: the name of the procedure has been defined before!";

errors.push\_back(line + error);

}

add\_func($2.value,"integer",func\_cur\_node);

$$.value = $2.value;

is\_main = 0;

}

;

formal\_parameter : LEFTCIRCLE parameter\_lists RIGHTCIRCLE

{

$$.value = "(" + $2.value + ")";

}

| LEFTCIRCLE parameter\_lists SEMICOLON RIGHTCIRCLE

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: unexpected semicolon!";

errors.push\_back(line + error);

$$.value = "(" + $2.value + ")";

}

|

{

$$.value = "";

}

;

parameter\_lists : parameter\_lists SEMICOLON parameter\_list

{

if ($3.value != "")

{

string parm = $3.value;

if ( parm.find("var+") == 0 )

{

parm = parm.substr(4);

}

add\_parameter(parm);

$$.value = $1.value + ", " + definition\_trans($3.value,true);

}

else

{

$$.value = $1.value;

}

}

| parameter\_lists error parameter\_list

{

$$.value = $1.value + ";" + $3.value;

string parm = $3.value;

if ( parm.find("var+") == 0 )

{

parm = parm.substr(4);

}

add\_parameter(parm);

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: semicolon missing, wrong separate symbol!";

errors.push\_back(line + error);

}

| parameter\_list

{

if ($1.value != "")

{

string parm = $1.value;

if ( parm.find("var+") == 0 )

{

parm = parm.substr(4);

}

add\_parameter(parm);

$$.value = definition\_trans($1.value,true);

}

}

;

parameter\_list : var\_parameter

{

$$.value = $1.value;

$$.is\_var = true;

}

| value\_parameter

{

$$.value = $1.value;

}

;

var\_parameter : VAR value\_parameter

{

$$.value = "var+" + $2.value;

}

;

value\_parameter : identifier\_list COLON type

{

if (check\_in\_ids($1.value) == 0 || check\_repeat\_definition($1.value) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable in parameters!";

errors.push\_back(line + error);

}

if ($1.value == "")

{

$$.value = "";

}

else

{

$$.value = $3.value + ":" + $1.value;

}

}

| identifier\_list

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: type missing!";

errors.push\_back(line + error);

if (check\_in\_ids($1.value) == 0 || check\_repeat\_definition($1.value) == 0)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: redefine the variable in parameters!";

errors.push\_back(line + error);

}

if ($1.value == "")

{

$$.value = "";

}

else

{

$$.value = "int:" + $1.value;

}

}

;

compound\_statement : BEGIN1 statement\_list END

{

$$.value = $2.value + "\n";

}

| BEGIN1 statement\_list SEMICOLON END

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: unexpected semicolon!";

errors.push\_back(line + error);

$$.value = $2.value + "\n";

}

;

statement\_list : statement\_list SEMICOLON statement

{

$$.value = $1.value + "\n\t" + $3.value;

}

| statement\_list statement

{

$$.value = $1.value + "\n\t" + $2.value;

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: semicolon missing, wrong separate symbol!";

errors.push\_back(line + error);

}

| statement

{

$$.value = $1.value;

}

;

statement : variable ASSIGNOP expression

{

if ($1.value == "return")

{

$$.value = "return " + $3.value;

}

if ( $1.value != "return" )

{

$$.value = $1.value + "=" + $3.value + ";";

}

check\_type\_error\_output($1.type,$3.type,$1.row\_no,$1.col\_no,$3.row\_no,$3.col\_no);

}

| variable RELOP expression

{

if ($1.value == "return")

{

$$.value = "return " + $3.value;

}

if ( $1.value != "return" )

{

$$.value = $1.value + "=" + $3.value + ";";

}

check\_type\_error\_output($1.type, $3.type, $1.row\_no, $1.col\_no, $3.row\_no, $3.col\_no);

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: assignrelop is wrong!";

errors.push\_back(line + error);

}

| variable error ASSIGNOP expression

{

if ($1.value == "return")

{

$$.value = "return " + $3.value;

}

else

{

$$.value = $1.value + "=" + $3.value + ";";

}

check\_type\_error\_output($1.type, $3.type, $1.row\_no, $1.col\_no, $3.row\_no, $3.col\_no);

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: too many symbols before '=' !";

errors.push\_back(line + error);

}

| call\_procedure\_statement

{

$$.value = $1.value + ";";

}

| compound\_statement

{

$$.value = $1.value;

}

| IF expression THEN statement else\_part

{

$$.value = "if(" + $2.value + ")" + "\n\t{\n\t\t" + $4.value + "\n\t}\n" + $5.value;

if (!($2.type == "bool" || $2.type == "constbool"))

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: the if expression is not bool!";

errors.push\_back(line + error);

}

}

| S case\_body END

{

$$.value = "switch(" + $1.value + "){\n" + $2.value + "}\n";

}

| S case\_body

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no+1) + " ";

string error = "error message: end is missing here!";

errors.push\_back(line + error);

$$.value = "switch(" + $1.value + "){\n" + $2.value + "}\n";

}

| WHILE expression DO statement

{

if (!($2.type == "bool" || $2.type == "constbool"))

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: the while expression is not bool!";

errors.push\_back(line + error);

}

$$.value = $1.value + "(" + $2.value + ")\n\t{\n\t\t" + $4.value + "\n\t}\n";

}

| WHILE expression statement

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: must have a do in the while statement!";

errors.push\_back(line + error);

if (!($2.type == "bool" || $2.type == "constbool"))

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: the while expression is not bool!";

errors.push\_back(line + error);

}

$$.value = $1.value + "(" + $2.value + ")\n\t{\n\t\t" + $3.value + "\n\t}\n";

}

| REPEAT statement\_list UNTIL expression

{

$$.value = "do{\n" + $2.value + "}while(" + $4.value + ")\n";

}

| T ASSIGNOP expression updown expression DO statement

{

check\_type\_error\_output($2.type, $4.type, $2.row\_no, $2.col\_no, $4.row\_no, $4.col\_no);

check\_type\_error\_output($2.type, $6.type, $2.row\_no, $2.col\_no, $6.row\_no, $6.col\_no);

if($4.value == "to")

{

$$.value = "for(" + $1.value + "=" + $3.value + ";" + $1.value + "<=" + $5.value + ";" + $1.value + "++){\n" + $6.value + "}\n";

}

else

{

$$.value = "for(" + $1.value + "=" + $3.value + ";" + $1.value + ">=" + $5.value + ";" + $1.value + "--){\n" + $6.value + "}\n";

}

}

| T ASSIGNOP expression updown expression statement

{

check\_type\_error\_output($1.type, $3.type, $1.row\_no, $1.col\_no, $3.row\_no, $3.col\_no);

check\_type\_error\_output($1.type, $5.type, $1.row\_no, $1.col\_no, $5.row\_no, $5.col\_no);

string line = "line " + int2str($5.row\_no) + " column " + int2str($5.col\_no+1) + " ";

string error = "error message: must have do in the for statement!";

errors.push\_back(line + error);

if($4.value == "to")

{

$$.value = "for(" + $1.value + "=" + $3.value + ";" + $1.value + "<=" + $5.value + ";" + $1.value + "++){\n" + $6.value + "}\n";

}

else

{

$$.value = "for(" + $1.value + "=" + $3.value + ";" + $1.value + ">=" + $5.value + ";" + $1.value + "--){\n" + $6.value + "}\n";

}

}

|

{

$$.value = "";

}

| write

{

$$.value = $1.value;

}

| read

{

$$.value = $1.value;

}

;

T : FOR ID

{

$$.value = $2.value;

$$.type = find\_local\_var\_func\_type($2.value,"");

}

;

write : WRITE LEFTCIRCLE expression\_list RIGHTCIRCLE

{

$$.value = write\_trans($3.value,$3.type,$3.row\_no,$3.col\_no);

dim = 0;

}

| WRITELN LEFTCIRCLE expression\_list RIGHTCIRCLE

{

$$.value = writeln\_trans($3.value,$3.type,$3.row\_no,$3.col\_no);

dim = 0;

}

| WRITELN

{

$$.value = "printf(\"\\n\")";

}

;

read : READ LEFTCIRCLE variable\_list RIGHTCIRCLE

{

$$.value = read\_trans($3.value,$3.type,$3.row\_no,$3.col\_no);

}

S : CASE expression OF

{

$$.type = $2.type;

cur\_case\_type = $2.type;

$$.value = $2.value;

}

;

expression : simple\_expression RELOP simple\_expression

{

if ( $1.type == "float" && $3.type == "float" || $1.type == "float" && $3.type == "int" || $1.type == "int" &&

$3.type == "int" || $1.type == "float" && $3.type == "int" || $1.type == "char" && $3.type == "char")

{

$$.type = "bool";

}

else if ($1.type == "bool" && $3.type == "bool")

{

if ($2.value == "=")

{

$$.type = "bool";

}

else

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: invalid relop between bools!";

errors.push\_back(line + error);

$$.type = "NULL";

}

}

else

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: invalid relop between expressions!";

errors.push\_back(line + error);

$$.type = "NULL";

}

$$.value = $1.value + $2.value + $3.value;

}

| simple\_expression

{

$$.value = $1.value;

$$.type = $1.type;

}

;

simple\_expression : term

{

$$.value = $1.value;

$$.type = $1.type;

}

| ADDOP term

{

if ( $2.type == "int" || $2.type == "float" )

$$.type = $2.type;

else

$$.type = "NULL";

$$.value = $1.value + $2.value;

}

| MINUSOP term

{

if ( $2.type == "int" || $2.type == "float" )

$$.type = $2.type;

else

$$.type = "NULL";

$$.value = $1.value + $2.value;

}

| simple\_expression ADDOP term

{

if ( $2.value == "or" )

{

$2.value = "||";

}

$$.value = $1.value + $2.value + $3.value;

$$.type = combine\_type($1.type, $3.type);

}

| simple\_expression MINUSOP term

{

$$.value = $1.value + $2.value + $3.value;

$$.type = combine\_type($1.type, $3.type);

}

;

term : term MULOP factor

{

if (($2.value == "/" || $2.value == "div" || $2.value == "mod")&& $3.value == "0")

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: the divisor can not be 0";

errors.push\_back(line + error);

}

if ( $2.value == "and" )

{

if ( !($1.type == "bool" || $1.type == "constbool") && ( $3.type == "bool" || $3.type == "constbool") )

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: type is not bool on either side of and operator";

errors.push\_back(line + error);

}

}

if ( $2.value == "mod" )

$2.value = "%";

if ( $2.value == "and" )

$2.value = "&&";

if ( $2.value == "div" )

$2.value = "/";

$$.value = $1.value + $2.value + $3.value;

$$.type = combine\_type($1.type, $3.type);

}

| factor

{

$$.value = $1.value;

$$.type = $1.type;

}

;

factor : ID LEFTCIRCLE expression\_list RIGHTCIRCLE

{

$$.value = $1.value + "(" + $3.value + ")";

if ( param\_type\_check($1.value, $3.type) != 0 )

{

$$.type = find\_func\_type($1.value);

if ( $$.type == "NULL" )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: function name not valid in the context!";

errors.push\_back(line + error);

}

}

else

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: check your parameter numbers or types!";

errors.push\_back(line + error);

$$.type = find\_func\_type($1.value);

if ( $$.type == "NULL" )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: function name not valid in the context!";

errors.push\_back(line + error);

}

}

dim = 0;

}

| ID LEFTCIRCLE RIGHTCIRCLE

{

$$.value = $1.value + "()";

$$.type = find\_func\_type($1.value);

if (call\_procedure\_id\_check($1.value)->symbol\_list->parameters == NULL)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no+1) + " ";

string error = "error message: unexpected circles!";

errors.push\_back(line + error);

}

else

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no+1) + " ";

string error = "error message: the parameters unmatched!";

errors.push\_back(line + error);

}

dim = 0;

}

| LEFTCIRCLE expression RIGHTCIRCLE

{

$$.value = "(" + $2.value + ")";

$$.type = $2.type;

}

| NOT factor

{

if ( $2.type == "bool" )

{

$$.value = "!" + $2.value;

$$.type = "bool";

}

else

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: factor not bool type";

errors.push\_back(line + error);

$$.value = "!" + $2.value;

$$.type = "NULL";

}

}

| unsigned\_const\_variable

{

$$.value = $1.value;

$$.type = $1.type;

}

| variable

{

$$.type = $1.type;

$$.value = $1.value;

}

;

unsigned\_const\_variable : FLOAT

{

$$.value = $1.value;

$$.type = "float";

}

| INT

{

$$.value = $1.value;

$$.type = "int";

}

| LETTER

{

$$.value = $1.value;

$$.type = "char";

}

;

variable : IDM id\_varparts

{

string name = $1.value;

if ( name.find("return") == 0 )

{

name = name.substr(6);

$1.value = $1.value.substr(0, 6);

}

string result = "";

if ( recover == 1 )

$$.type = find\_local\_var\_func\_type(name, "array"+$2.value);

if ( recover == 2 )

$$.type = find\_local\_var\_func\_type(name, "struct"+$2.value);

if ( recover == 0 )

$$.type = find\_local\_var\_func\_type(name, $2.value);

recover = 0;

if ( $2.value != "" )

{

if ( $2.value.find("[") != -1 )

{

$2.value = $2.value.substr(1);

$2.value = $2.value.substr(0,$2.value.length() - 1);

vector<string> cons = split($2.value,",");

for (int i = 0; i < cons.size(); i ++)

{

result = result + "[" + cons[i] + "]";

}

}

else if ( $2.value.find(".") != -1)

{

result = $2.value;

}

}

$$.value = $1.value + result;

}

;

IDM : ID

{

if (func\_cur\_node->name == $1.value)

{

$$.value = "return" + $1.value;

}

else

{

$$.value = $1.value;

}

array\_id = $1.value;

}

;

id\_varparts : id\_varparts id\_varpart

{

if ($2.value.find("array") == 0)

{

$2.value = $2.value.substr(5);

recover = 1;

}

if ($2.value.find("struct") == 0)

{

$2.value = "." + $2.value.substr(6);

recover = 2;

}

$$.value = $1.value + $2.value;

}

|

{

$$.value = "";

}

;

id\_varpart : LEFTCUBE expression\_list RIGHTCUBE

{

int valid = 1;

string type = find\_local\_var\_func\_type(array\_id, "");

vector<string> newnums;

if ( type.find("array") == 0 )

{

int pos = type.find("[");

type = type.substr(pos+1);

type = type.erase(type.length()-1, 1);

vector<string> dimns = split(type, ",");

vector<string> nums = split($2.value, ",");

if ( nums.size() == dimns.size() )

{

for ( int i = 0; i < dimns.size() && i < nums.size(); i++ )

{

vector<string> bounds = split(dimns[i],"|");

int low = atoi(bounds[0].data());

int up = atoi(bounds[1].data());

int num = atoi(nums[i].data());

if ( num < low || num > up )

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: out of bounds in the array!";

errors.push\_back(line + error);

}

newnums.push\_back(int2str(num-low));

}

}

if ( dim != dimns.size() ) valid = 0;

if ( valid == 0 )

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: dimension errors in the array!";

errors.push\_back(line + error);

for (int i = 0; i < nums.size(); i ++)

{

newnums.push\_back(nums[i]);

}

}

}

string new\_exp = "";

for (int i = 0; i < newnums.size(); i ++)

{

new\_exp = new\_exp + newnums[i] + ",";

}

new\_exp = new\_exp.substr(0,new\_exp.size() - 1);

$$.value = "array[" + new\_exp + "]";

dim = 0;

}

| PROGRAMEND ID

{

$$.value = "struct" + $2.value;

}

;

variable\_list : variable\_list COMMA variable

{

$$.value = $1.value + "," + $3.value;

$$.type = $1.type + "\*" + $3.type;

}

| variable

{

$$.value = $1.value;

$$.type = $1.type;

}

;

else\_part : ELSE statement

{

$$.value = "\telse\n\t{\n\t\t" + $2.value + "\n\t}\n";

}

|

{

$$.value = "";

}

;

case\_body : branch\_list

{

$$.value = $1.value;

branches.clear();

}

|

{

$$.value = "";

}

;

branch\_list : branch\_list SEMICOLON branch

{

$$.value = $1.value + $3.value;

}

| branch

{

$$.value = $1.value;

}

;

branch : const\_list COLON statement

{

vector<string> cases;

int error1 = 0;

$$.value = "";

cases = split($1.value,",");

for ( int i = 0; i < cases.size(); i++ )

{

if( nums.size() == 0 && branches.size() == 0 )

{

nums.push\_back(cases[i]);

branches.push\_back($3.value);

$$.value = $$.value + "case " + cases[i] + ":" + $3.value + "break;\n";

}

else

{

for ( int j = 0; j < nums.size(); j++ )

{

if ( nums[j] == cases[i] )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: branches conflict!";

errors.push\_back(line + error);

error1 = 1;

}

}

if ( error1 == 0 )

{

nums.push\_back(cases[i]);

branches.push\_back($3.value);

$$.value = $$.value + "case " + cases[i] + ":" + $3.value + "break;\n";

}

}

error1 = 0;

}

}

;

const\_list : const\_list COMMA const\_variable

{

int error = 0;

if ( $3.type != cur\_case\_type )

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: const type not matched with case type!";

errors.push\_back(line + error);

}

$$.value = $1.value + "," + $3.value;

}

| const\_variable

{

int error = 0;

if ( $1.type != cur\_case\_type )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: const type not matched with case type!";

errors.push\_back(line + error);

}

$$.value = $1.value;

}

;

updown : TO

{

$$.value = $1.value;

}

| DOWNTO

{

$$.value = $1.value;

}

;

call\_procedure\_statement : ID

{

if ( call\_procedure\_id\_check($1.value) != NULL )

{

$$.type = find\_func\_type($1.value);

if ( $$.type != "void")

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: can not call a function directly in a statement!";

errors.push\_back(line + error);

}

}

else

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: the function is not found!";

errors.push\_back(line + error);

$$.type = "NULL";

}

$$.value = $1.value + "()";

}

| ID LEFTCIRCLE expression\_list RIGHTCIRCLE

{

$$.value = $1.value + "(" + $3.value + ")";

if ( param\_type\_check($1.value, $3.type) == 1 )

{

$$.type = find\_func\_type($1.value);

if ( $$.type == "NULL" )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: function name not valid in the context!";

errors.push\_back(line + error);

}

}

else if ( param\_type\_check($1.value, $3.type) == -1 )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: the function does not found!";

errors.push\_back(line + error);

$$.type = "NULL";

}

else

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: the parameters unmatched!";

errors.push\_back(line + error);

$$.type = "NULL";

}

dim = 0;

}

| ID expression\_list RIGHTCIRCLE

{

$$.value = $1.value + "(" + $2.value + ")";

if ( param\_type\_check($1.value, $3.type) == 1 )

{

$$.type = find\_func\_type($1.value);

if ( $$.type == "NULL" )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: function name not valid in the context!";

errors.push\_back(line + error);

}

}

else if ( param\_type\_check($1.value, $3.type) == -1 )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: the function does not found!";

errors.push\_back(line + error);

$$.type = "NULL";

}

else

{

string line = "line " + int2str($2.row\_no) + " column " + int2str($2.col\_no) + " ";

string error = "error message: the parameters unmatched!";

errors.push\_back(line + error);

$$.type = "NULL";

}

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no+1) + " ";

string error = "error message: leftcircle is missing!";

errors.push\_back(line + error);

dim = 0;

}

| ID LEFTCIRCLE expression\_list

{

$$.value = $1.value + "(" + $3.value + ")";

if ( param\_type\_check($1.value, $3.type) == 1 )

{

$$.type = find\_func\_type($1.value);

if ( $$.type == "NULL" )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: function name not valid in the context!";

errors.push\_back(line + error);

}

}

else if ( param\_type\_check($1.value, $3.type) == -1 )

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no) + " ";

string error = "error message: the function does not found!";

errors.push\_back(line + error);

$$.type = "NULL";

}

else

{

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no) + " ";

string error = "error message: the parameters unmatched!";

errors.push\_back(line + error);

$$.type = "NULL";

}

string line = "line " + int2str($3.row\_no) + " column " + int2str($3.col\_no+1) + " ";

string error = "error message: rightcircle is missing!";

errors.push\_back(line + error);

dim = 0;

}

| ID LEFTCIRCLE RIGHTCIRCLE

{

$$.value = $1.value + "()";

if (call\_procedure\_id\_check($1.value)->symbol\_list->parameters == NULL)

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no+1) + " ";

string error = "error message: unexpected circles!";

errors.push\_back(line + error);

}

else

{

string line = "line " + int2str($1.row\_no) + " column " + int2str($1.col\_no+1) + " ";

string error = "error message: the parameters unmatched!";

errors.push\_back(line + error);

}

dim = 0;

}

;

expression\_list : expression\_list COMMA expression

{

$$.value = $1.value + "," + $3.value;

$$.type = $1.type + "\*" + $3.type;

dim ++;

}

| expression

{

$$.value = $1.value;

$$.type = $1.type;

dim++;

}

;

%%

//赋值语句类型检查结果反馈

void check\_type\_error\_output(string type1, string type2, int rownum1, int colnum1, int rownum2, int colnum2)

{

string line;

string error;

if (assign\_type\_check(type1,type2) == 0) //错误：赋值语句左边为常量

{

line = "line " + int2str(rownum1) + " column " + int2str(colnum1) + " ";

error = "error message: the const variable can't be assigned!";

errors.push\_back(line + error);

}

if (assign\_type\_check(type1,type2) == -1) //错误：左部变量未定义或表达式类型无效

{

line = "line " + int2str(rownum1) + " column " + int2str(colnum1) + " ";

error = "error message: the variable has not been defined!";

errors.push\_back(line + error);

}

else if (assign\_type\_check(type1,type2) == -2) //错误：右部变量未定义或表达式类型无效

{

line = "line " + int2str(rownum2) + " column " + int2str(colnum2) + " ";

error = "error message: the variable has not been defined!";

errors.push\_back(line + error);

}

else if (assign\_type\_check(type1,type2) == -3) //错误：两侧符号均未定义或表达式类型无效

{

line = "line " + int2str(rownum1) + " column " + int2str(colnum1) + " ";

error = "error message: the variable has not been defined!";

errors.push\_back(line + error);

line = "line " + int2str(colnum2) + " column " + int2str(colnum2) + " ";

error = "error message: the variable has not been defined!";

errors.push\_back(line + error);

}

else if (assign\_type\_check(type1,type2) == -4) //错误：左右表达式类型有效但不匹配

{

line = "line " + int2str(rownum2) + " column " + int2str(colnum2) + " ";

error = "error message: type unmatched in the assignment!";

error = error + "\nleft\_type is " + type1 + " right\_type is " + type2;

errors.push\_back(line + error);

}

}

//write语句的翻译

string write\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no)

{

vector<string> exp = split(exp\_type, "\*");

vector<string> exp\_list = split(exp\_value, ",");

vector<string>::iterator it;

string str = "\"";

for (int i = 0; i < exp.size(); i ++)

{

if (exp[i] != "NULL")

{

if (exp[i] == "int" || exp[i] == "constint") //整型变量

{

str += "%d";

}

else if (exp[i] == "char" || exp[i] == "constchar") //字符型

{

str += "%c";

}

else if (exp[i] == "float" || exp[i] == "constfloat") //浮点型

{

str += "%f";

}

else if (exp[i] == "bool" || exp[i] == "constbool") //布尔型变量

{

it = exp\_list.begin() + i;

exp\_list.erase(it);

}

}

else

{

it = exp\_list.begin() + i;

exp\_list.erase(it);

}

}

str += "\"";

for (int i = 0; i < exp\_list.size(); i ++)

{

str = str + "," + exp\_list[i];

}

return "printf(" + str + ");";

}

//writeln语句的翻译

string writeln\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no)

{

vector<string> exp = split(exp\_type, "\*");

vector<string> exp\_list = split(exp\_value, ",");

vector<string>::iterator it;

string str = "\"";

for (int i = 0; i < exp.size(); i ++)

{

if (exp[i] != "NULL")

{

if (exp[i] == "int" || exp[i] == "constint")

{

str += "%d";

}

else if (exp[i] == "char" || exp[i] == "constchar")

{

str += "%c";

}

else if (exp[i] == "float" || exp[i] == "constfloat")

{

str += "%f";

}

else if (exp[i] == "bool" || exp[i] == "constbool")

{

it = exp\_list.begin() + i;

exp\_list.erase(it);

}

}

else

{

it = exp\_list.begin() + i;

exp\_list.erase(it);

}

}

str += "\\n\"";

for (int i = 0; i < exp\_list.size(); i ++)

{

str = str + "," + exp\_list[i];

}

return "printf(" + str + ");";

}

//read语句翻译

string read\_trans(string exp\_value,string exp\_type,int row\_no,int col\_no)

{

vector<string> exp = split(exp\_type, "\*");

vector<string> exp\_list = split(exp\_value, ",");

vector<string>::iterator it;

string str = "\"";

for (int i = 0; i < exp.size(); i ++)

{

if (exp[i] != "NULL")

{

if (exp[i] == "int")

{

str += "%d";

}

else if (exp[i] == "char")

{

str += "%c";

}

else if (exp[i] == "float")

{

str += "%f";

}

else

{

it = exp\_list.begin() + i;

exp\_list.erase(it);

}

}

else

{

it = exp\_list.begin() + i;

exp\_list.erase(it);

}

}

str += "\"";

for (int i = 0; i < exp\_list.size(); i ++)

{

str = str + ",&" + exp\_list[i];

}

return "scanf(" + str + ");";

}

//发生无法恢复的错误时（没有产生式可以规约），调用yyerror函数报错

void yyerror(char \*s) //当yacc遇到语法错误时，会回调yyerror函数，并且把错误信息放在参数s中

{

cerr << s << endl;//直接输出错误信息

}

//向函数表中加入一个节点表示当前处理的（子）函数

void add\_func(string name, string type, struct function \*currentNode)

{

struct function \*funcNode;

if (currentNode == NULL) //主函数作为头结点

{

currentNode = new struct function;

currentNode->next\_function = NULL;

func\_head\_node = currentNode;

}

else

{

funcNode = new struct function;

currentNode->next\_function = funcNode;

currentNode = funcNode;

}

currentNode->name = name;

currentNode->ret\_type = type;

currentNode->type\_num = 0;

currentNode->const\_num = 0;

currentNode->type\_start = -1;

currentNode->const\_start = -1;

currentNode->symbol\_list = NULL;

currentNode -> next\_function = NULL;

func\_cur\_node = currentNode;

}

//切分函数

vector<string> split(string &str,string pattern)

{

int pos;

vector<string> result;

if (str == "")

{

return result;

}

str += pattern;

int size = str.size();

for (int i = 0; i < size; i ++)

{

pos = str.find(pattern,i);

if (pos < size)

{

string tmp = str.substr(i,pos-i);

result.push\_back(tmp);

i = pos + pattern.size() - 1;

}

}

str = str.substr(0,size - pattern.length());

return result;

}

//去空格函数

string del\_space(string str)

{

string result = "";

for (int i = 0; i < str.size(); i ++)

{

if (str[i] != ' ')

{

result += str[i];

}

}

return result;

}

//函数返回值回写函数，先创建函数表节点，参数确定后再回填其返回类型

void redefine\_type( string type )

{

func\_cur\_node -> ret\_type = type;

}

//函数调用有效性检查函数

struct function\* call\_procedure\_id\_check( string id )

{

struct function\* func\_node = func\_head\_node;

if (is\_main == 1)

{

while( func\_node != NULL )

{

if ( func\_node -> name == id )

{

return func\_node;

}

func\_node = func\_node -> next\_function;

}

return NULL;

}

else

{

while( func\_node != NULL )

{

if ( func\_node -> name == id && func\_node != func\_cur\_node || func\_node == func\_cur\_node )

{

return func\_node;

}

func\_node = func\_node -> next\_function;

}

return NULL;

}

}

//调用函数时的参数检查，0：参数不匹配；1：参数匹配；-1：函数未找到

int param\_type\_check ( string id, string params )

//

{

struct function\* func\_node;

vector<string> param\_type = split(params, "\*");

int i = 0;

func\_node = call\_procedure\_id\_check(id);

string type;

if ( func\_node == NULL )

{

return -1;

}

else if ( func\_node != NULL )

{

struct symbol\* prms = func\_node -> symbol\_list -> parameters;

if ( prms != NULL )

{

while ( prms-> name != "" && params != "" && i < param\_type.size() )

{

if ( prms -> var\_type == "struct" )

{

type = "";

type = type + "struct" + "{";

struct symbol\* members = prms -> members;

while(members != NULL )

{

type += members -> name + ":" + members->var\_type;

if ( members -> next\_symbol != NULL )

{

type += ";";

}

members = members -> next\_symbol;

}

type += "}";

}

if ( prms -> var\_type == "array" )

{

type = "";

type = type + "array" + prms->array\_type + "[";

vector<struct array\_elem> members(prms -> bounds);

for ( int i = 0; i < members.size(); i++ )

{

string low\_bound;

string up\_bound;

low\_bound = int2str(members[i].low\_bound);

up\_bound = int2str(members[i].up\_bound);

type += low\_bound + "|" + up\_bound;

if ( i != members.size() - 1 )

type += ",";

}

type += "]";

}

if ( prms -> var\_type == "int" || prms -> var\_type == "float" || prms -> var\_type == "char" || prms -> var\_type == "bool")

{

type = prms -> var\_type;

}

if ( param\_type[i] != type )

return 0;

prms = prms -> next\_symbol;

i++;

}

if ( i != param\_type.size() || prms -> name != "" )

return 0;

return 1;

}

if ( prms == NULL && params == "" )

{

return 1;

}

}

return 0;

}

//调用函数时获取函数的返回类型

string find\_func\_type ( string id )

{

struct function\* func\_node = func\_head\_node;

while( func\_node != NULL )

{

if ( func\_node->name == id )

{

return func\_node -> ret\_type;

}

func\_node = func\_node -> next\_function;

}

return "NULL";

}

//向符号表的参数域增加参数

void add\_parameter(string parameters)

{

string new\_para;

string type;

string paras;

vector<string> para;

vector<string> member\_para;

if ( func\_cur\_node -> symbol\_list == NULL) //当前函数表节点没有参数节点，则创建

{

func\_cur\_node->symbol\_list = new struct symbol;

func\_cur\_node->symbol\_list->name = "";

if ( parameters == "" )

func\_cur\_node -> symbol\_list -> parameters = NULL;

else

{

func\_cur\_node -> symbol\_list -> parameters = new struct symbol;

param\_cur\_node = func\_cur\_node -> symbol\_list -> parameters;

}

}

if (parameters != "") //参数不为空，则添加到该函数表节点的第一个符号表节点的parameter域

new\_para = del\_space(parameters);

else new\_para = "";

if ( new\_para != "" )

{

int type\_start = new\_para.find(":");

type = new\_para.substr(0, type\_start);

if ( type\_start != -1 )

{

paras = new\_para.substr(type\_start+1);

para = split(paras, ",");

if ( type.find("struct") == 0 ) //参数类型为记录类型

{

for ( int k = 0; k < para.size(); k++ )

{

param\_cur\_node -> name = para[k];

param\_cur\_node -> var\_type = "struct";

param\_cur\_node -> members = new struct symbol;

struct symbol\* pointer = param\_cur\_node -> members;

pointer-> next\_symbol = NULL;

string temp = type.substr(type.find("{")+1);

string members = temp.erase(temp.find("}"));

member\_para = split(members,";");

for ( int i = 0; i < member\_para.size(); i++ )

{

vector<string> tmp = split(member\_para[i],":");

vector<string> vars = split(tmp[0],",");

for ( int j = 0; j < vars.size(); j++ ) //存储每个成员的信息

{

pointer -> name = vars[j];

pointer -> var\_type = tmp[1];

struct symbol \*tmp = new struct symbol;

if ( i == member\_para.size()-1 && j == vars.size()-1 ){

pointer -> next\_symbol = NULL;

}

else{

pointer -> next\_symbol = tmp;

pointer = tmp;

}

}

}

param\_cur\_node -> next\_symbol = new struct symbol;

param\_cur\_node = param\_cur\_node -> next\_symbol;

param\_cur\_node -> name == "";

}

}

else if( type.find("array") == 0 ) //参数类型为数组类型

{

type = type.substr(5);

for ( int k = 0; k < para.size(); k++ )

{

vector<string> bnds;

param\_cur\_node -> name = para[k];

param\_cur\_node -> var\_type = "array";

param\_cur\_node -> array\_type = type.substr(0, type.find("["));

struct array\_elem elem;

string temp = type.substr(type.find("[")+1);

string members = temp.erase(temp.find("]"), temp.size()-1 );

bnds = split(members, ",");

for ( int s = 0; s < bnds.size(); s ++ )

{

vector<string> bnd = split(bnds[s], "|");

elem.low\_bound = atoi(bnd[0].data());

elem.up\_bound = atoi(bnd[1].data());

(param\_cur\_node -> bounds).push\_back(elem);

}

param\_cur\_node -> next\_symbol = new struct symbol;

param\_cur\_node = param\_cur\_node -> next\_symbol;

param\_cur\_node -> name == "";

}

}

else if( type.find("int") == 0 || type.find("float") == 0 || type.find("bool") == 0 || type.find("char") == 0)//参数类型为简单变量类型

{

for ( int k = 0; k < para.size(); k++ )

{

param\_cur\_node -> name = para[k];

param\_cur\_node -> var\_type = type;

param\_cur\_node -> next\_symbol = new struct symbol;

param\_cur\_node = param\_cur\_node -> next\_symbol;

param\_cur\_node -> name == "";

}

}

}

}

sym\_cur\_node = func\_cur\_node->symbol\_list;

sym\_cur\_node -> next\_symbol = NULL;

}

//向符号表中加入一个符号，记录其名字和类型，若为记录类型则要保存成员信息；若为数组类型需要保存上下界和维数

void add\_symbol(string symbols, string type)

{

string new\_sym = del\_space(symbols);

vector<string> syms = split(new\_sym, ",");

struct symbol \*symbol\_cur = new struct symbol;

vector<string> member\_para;

sym\_cur\_node->next\_symbol = symbol\_cur;

symbol\_cur -> parameters = NULL;

symbol\_cur->next\_symbol = NULL;

if ( func\_head\_node == func\_cur\_node && time\_r == 1 ) //主函数中需要创建一个空的参数节点，因为主函数参数在转换中没有意义

{

add\_parameter("");

time\_r = 0;

}

if ( type.find("struct") == 0 ) //增加记录类型的变量

{

for ( int k = 0; k < syms.size(); k++ )

{

symbol\_cur -> name = syms[k];

symbol\_cur -> var\_type = "struct";

symbol\_cur -> members = new struct symbol;

struct symbol\* pointer = symbol\_cur -> members;

pointer-> next\_symbol = NULL;

string temp = type.substr(type.find("{")+1);

string members = temp.erase(temp.find("}"));

member\_para = split(members,";");

for ( int i = 0; i < member\_para.size(); i++ ) //保存成员信息

{

vector<string> tmp = split(member\_para[i]," ");

vector<string> vars = split(tmp[1],",");

for ( int j = 0; j < vars.size(); j++ )

{

pointer -> name = vars[j];

pointer -> var\_type = tmp[0];

struct symbol \*tmp = new struct symbol;

if ( i == member\_para.size()-1 && j == vars.size()-1 ){

pointer -> next\_symbol = NULL;

}

else{

pointer -> next\_symbol = tmp;

pointer = tmp;

}

}

}

if ( k == syms.size() - 1)

symbol\_cur -> next\_symbol = NULL;

else

{

symbol\_cur -> next\_symbol = new struct symbol;

symbol\_cur = symbol\_cur -> next\_symbol;

}

}

}

else if(type.find("array") == 0) //增加数组变量

{

type = type.substr(5);

for ( int k = 0; k < syms.size(); k++ )

{

vector<string> bnds;

symbol\_cur -> name = syms[k];

symbol\_cur -> var\_type = "array";

symbol\_cur -> array\_type = type.substr(0, type.find("["));

struct array\_elem elem;

string temp = type.substr(type.find("[")+1);

string members = temp.erase(temp.find("]"), temp.size()-1 );

bnds = split(members, ",");

for ( int s = 0; s < bnds.size(); s ++ )//记录维数和上下界

{

vector<string> bnd = split(bnds[s], "|");

elem.low\_bound = atoi(bnd[0].data());

elem.up\_bound = atoi(bnd[1].data());

symbol\_cur -> bounds.push\_back(elem);

}

if ( k == syms.size() - 1)

symbol\_cur -> next\_symbol = NULL;

else

{

symbol\_cur -> next\_symbol = new struct symbol;

symbol\_cur = symbol\_cur -> next\_symbol;

}

}

}

else if (type.find("int") == 0 || type.find("float") == 0 || type.find("char") == 0 || type.find("bool") == 0)//保存普通类型变量

{

for ( int k = 0; k < syms.size(); k++ )

{

symbol\_cur -> name = syms[k];

symbol\_cur -> var\_type = type;

if ( k == syms.size() - 1)

symbol\_cur -> next\_symbol = NULL;

else{

symbol\_cur -> next\_symbol = new struct symbol;

symbol\_cur = symbol\_cur -> next\_symbol;

}

}

}

sym\_cur\_node = symbol\_cur;

}

//赋值类型检查

int assign\_type\_check( string left\_type, string right\_type )

{

if ( left\_type == "NULL" && right\_type != "NULL")

{

return -1;

}

else if (left\_type != "NULL" && right\_type == "NULL")

{

return -2;

}

else if (left\_type == "NULL" && right\_type == "NULL")

{

return -3;

}

else if ( left\_type.find("const") == 0 )

{

return 0;

}

else if ( left\_type.find("int") == 0 && right\_type.find("constint") == 0 || left\_type.find("int") == 0 && right\_type.find("int") == 0)

{

return 1;

}

else if ( left\_type.find("array") != -1 && right\_type.find("array") != -1 && left\_type == right\_type )

{

return 1;

}

else if( left\_type == "float" && right\_type == "int" || left\_type == "float" && right\_type == "constint"

|| left\_type.find("float") == 0 && right\_type.find("constfloat") == 0)

{

return 1;

}

else if ( left\_type == "char" && right\_type == "char" || left\_type.find("char") == 0 && right\_type.find("constchar") == 0 )

{

return 1;

}

else if ( left\_type.find("bool") == 0 && right\_type.find("bool") == 0 || left\_type.find("bool") == 0 && right\_type.find("constbool") == 0 )

{

return 1;

}

else if ( left\_type.find("struct") != -1 && right\_type.find("struct") != -1 && left\_type == right\_type )

{

return 1;

}

else

{

return -4;

}

}

//将运算符号两边的变量类型合并

string combine\_type( string type1, string type2 )//only int and float

{

if( type1 == "int" && type2 == "int" || type1 == "constint" && type2 == "int"

|| type2 == "constint" && type1 == "int" || type1 == "constint" && type2 == "constint")

{

return "int";

}

else if ( type1 == "char" && type2 == "char" || type1 == "constchar" && type2 == "char"

|| type1 == "char" && type2 == "constchar" || type1 == "constchar" && type2 == "constchar")

{

return "char";

}

else if ( type1 == "float" && type2 == "int" || type1 == "float" && type2 == "constint"

|| type1 == "float" && type2 == "float" || type1 == "float" && type2 == "constfloat"

|| type1 == "constfloat" && type2 == "int" || type1 == "constfloat" && type2 == "constint"

|| type1 == "constfloat" && type2 == "float" || type1 == "constfloat" && type2 == "constfloat"

|| type1 == "int" && type2 == "float" || type1 == "int" && type2 == "constfloat"

|| type1 == "constint" && type2 == "float" || type1 == "constint" && type2 == "constfloat")

{

return "float";

}

else if ( type1 == "bool" && type2 == "bool" || type1 == "bool" && type2 == "constbool"

|| type1 == "constbool" && type2 == "bool" || type1 == "constbool" && type2 == "constbool")

{

return "bool";

}

else

{

return "NULL";

}

}

//整型转字符串函数

string int2str( int int\_temp )

{

string string\_temp;

stringstream stream;

stream<<int\_temp;

string\_temp=stream.str(); //此处也可以用 stream>>string\_temp

return string\_temp;

}

//获得局部变量或者当前所在函数的类型

string find\_local\_var\_func\_type( string id, string id\_varparts )//only parameter node

{

struct symbol\* sym\_node;

if ( is\_main == 1 )

{

sym\_node = func\_head\_node -> symbol\_list;

}

else

{

sym\_node = func\_cur\_node -> symbol\_list;

}

if ( id == func\_cur\_node -> name )

return func\_cur\_node -> ret\_type;

if ( id\_varparts == "" )

{

int start = func\_cur\_node -> const\_start;

int num = func\_cur\_node -> const\_num;

if ( num != 0 )

{

for ( int i = start; i < start + num; i++ )

{

if ( const\_table[i].name == id )

return "const"+const\_table[i].const\_type;

}

}

}

while(sym\_node != NULL)

{

if( sym\_node -> parameters != NULL )

{

struct symbol\* param\_node = sym\_node -> parameters;

while( param\_node != NULL )

{

if( param\_node -> name == id )

{

if ( id\_varparts.find("array") != -1 )

{

return param\_node -> array\_type;

}

else if ( id\_varparts.find("struct") != -1 )

{

string member = id\_varparts.substr(7);

struct symbol\* mems = param\_node -> members;

while( mems != NULL )

{

if ( mems -> name == member )

{

return mems -> var\_type;

}

mems = mems -> next\_symbol;

}

}

else if ( id\_varparts == "" )

{

if( param\_node -> var\_type == "array" )

{

string arraytp = "";

arraytp = arraytp + "array" + param\_node -> array\_type + "[";

vector<struct array\_elem> bnds(param\_node -> bounds);

for ( int i = 0; i < bnds.size(); i++ )

{

string low\_bound;

string up\_bound;

low\_bound = int2str(bnds[i].low\_bound);

up\_bound = int2str(bnds[i].up\_bound);

arraytp = arraytp + low\_bound + "|" + up\_bound;

if ( i != bnds.size() - 1 )

arraytp = arraytp + ",";

}

arraytp = arraytp + "]";

return arraytp;

}

else if ( param\_node -> var\_type == "struct" )

{

string structtp = "";

structtp = structtp + "struct" + "{";

struct symbol\* members = param\_node -> members;

while(members != NULL )

{

structtp += members -> name + ":" + members->var\_type;

if ( members -> next\_symbol != NULL )

{

structtp += ";";

}

members = members -> next\_symbol;

}

structtp += "}";

return structtp;

}

}

return param\_node -> var\_type;

}

param\_node = param\_node -> next\_symbol;

}

}

struct symbol\* cur\_node = sym\_node;

if ( cur\_node -> name == id )

{

if ( id\_varparts.find("array") != -1 )

{

return cur\_node -> array\_type;

}

else if( id\_varparts.find("struct") != -1 )

{

string member = id\_varparts.substr(7);

struct symbol\* mems = cur\_node -> members;

while( mems != NULL )

{

if ( mems -> name == member )

{

return mems -> var\_type;

}

mems = mems -> next\_symbol;

}

}

else if ( id\_varparts == "" )

{

if( cur\_node -> var\_type == "array" )

{

string arraytp = "";

arraytp = arraytp + "array" + cur\_node -> array\_type + "[";

vector<struct array\_elem> bnds(cur\_node -> bounds);

for ( int i = 0; i < bnds.size(); i++ )

{

string low\_bound;

string up\_bound;

low\_bound = int2str(bnds[i].low\_bound);

up\_bound = int2str(bnds[i].up\_bound);

arraytp = arraytp + low\_bound + "|" + up\_bound;

if ( i != bnds.size() - 1 )

arraytp = arraytp + ",";

}

arraytp = arraytp + "]";

return arraytp;

}

else if ( cur\_node -> var\_type == "struct" )

{

string structtp = "";

structtp = structtp + "struct" + "{";

struct symbol\* members = cur\_node -> members;

while(members != NULL )

{

structtp += members -> name + ":" + members->var\_type;

if ( members -> next\_symbol != NULL )

{

structtp += ";";

}

members = members -> next\_symbol;

}

structtp += "}";

return structtp;

}

}

return cur\_node -> var\_type;

}

sym\_node = sym\_node -> next\_symbol;

}

return find\_global\_var\_type(id, id\_varparts);

}

//获取全局变量类型

string find\_global\_var\_type( string id, string id\_varparts )

{

struct symbol \*sym\_node = func\_head\_node -> symbol\_list -> next\_symbol;

if ( id\_varparts == "" )

{

int start = func\_head\_node -> const\_start;

int num = func\_head\_node -> const\_num;

if ( num != 0 )

{

for ( int i = start; i < start + num; i++ )

{

if ( const\_table[i].name == id )

return "const"+const\_table[i].const\_type;

}

}

}

while (sym\_node != NULL)

{

if ( sym\_node -> name == id )

{

if ( id\_varparts.find("array") != -1 )

{

return sym\_node -> array\_type;

}

else if( id\_varparts.find("struct") != -1 )

{

string member = id\_varparts.substr(7);

struct symbol\* mems = sym\_node -> members;

while( mems != NULL )

{

if ( mems -> name == member )

{

return mems -> var\_type;

}

mems = mems -> next\_symbol;

}

}

else if ( id\_varparts == "" )

{

if( sym\_node -> var\_type == "array" )

{

string arraytp = "";

arraytp = arraytp + "array" + sym\_node -> array\_type + "[";

vector<struct array\_elem> bnds(sym\_node -> bounds);

for ( int i = 0; i < bnds.size(); i++ )

{

string low\_bound;

string up\_bound;

low\_bound = int2str(bnds[i].low\_bound);

up\_bound = int2str(bnds[i].up\_bound);

arraytp = arraytp + low\_bound + "|" + up\_bound;

if ( i != bnds.size() - 1 )

arraytp = arraytp + ",";

}

arraytp = arraytp + "]";

return arraytp;

}

else if ( sym\_node -> var\_type == "struct" )

{

string structtp = "";

structtp = structtp + "struct" + "{";

struct symbol\* members = sym\_node -> members;

while(members != NULL )

{

structtp += members -> name + ":" + members->var\_type;

if ( members -> next\_symbol != NULL )

{

structtp += ";";

}

members = members -> next\_symbol;

}

structtp += "}";

return structtp;

}

}

return sym\_node -> var\_type;

}

sym\_node = sym\_node -> next\_symbol;

}

return "NULL";

}

//获取声明常量的类型

string get\_const\_type(string id)

{

int start = func\_cur\_node->const\_start;

int num = func\_cur\_node->const\_num;

string result = "not exist";

if (num == 0)

{

return result;

}

for (int i = start; i < start + num; i ++)

{

if (const\_table[i].name == id)

{

result = const\_table[i].const\_type;

}

}

start = func\_head\_node->const\_start;

num = func\_head\_node->const\_num;

for (int j = start; j < num + start; j ++)

{

if (const\_table[j].name == id)

{

result = const\_table[j].const\_type;

}

}

return result;

}

//获取声明常量的名字

string get\_const\_value(string id)

{

int start = func\_cur\_node->const\_start;

int num = func\_cur\_node->const\_num;

string result = "not exist";

if (num == 0)

{

return result;

}

for (int i = start; i < start + num; i ++)

{

if (const\_table[i].name == id)

{

return const\_table[i].name;

}

}

start = func\_head\_node->const\_start;

num = func\_head\_node->const\_num;

for (int j = start; j < num + start; j ++)

{

if (const\_table[j].name == id)

{

return const\_table[j].name;

}

}

return result;

}

//从type\_table中获取type类型，找不到返回not exist

string get\_typedef\_type(string id)

{

int start = func\_cur\_node->type\_start;

int num = func\_cur\_node->type\_num;

string result = "not exist";

if (num == 0)

{

return result;

}

for (int i = start; i < num + start; i ++)

{

if (type\_table[i].name == id)

{

return type\_table[i].type;

}

}

start = func\_head\_node->type\_start;

num = func\_head\_node->type\_num;

for (int j = start; j < num + start; j ++)

{

if (type\_table[j].name == id)

{

return type\_table[j].type;

}

}

return result;

}

//从type\_table中获取type名，找不到返回not exist

string get\_typedef\_value(string id)

{

int start = func\_cur\_node->type\_start;

int num = func\_cur\_node->type\_num;

string result = "not exist";

if (num == 0)

{

return result;

}

for (int i = start; i < num + start; i ++)

{

if (type\_table[i].name == id)

{

result = type\_table[i].name;

}

}

start = func\_head\_node->type\_start;

num = func\_head\_node->type\_num;

for (int j = start; j < num + start; j ++)

{

if (type\_table[j].name == id)

{

result = type\_table[j].name;

}

}

return result;

}

//在id\_list中直接进行重复检查，重复返回0，不重复返回1

int check\_in\_ids(string &id\_list)

{

string new\_list;

vector<string> ids;

if (id\_list != "")

{

new\_list = del\_space(id\_list);

}

else

{

new\_list = "";

}

ids = split(new\_list,",");

int size1 = ids.size();

sort(ids.begin(),ids.end());

ids.erase( unique(ids.begin(),ids.end()), ids.end() );

int size2 = ids.size();

id\_list = "";

for (int i = 0; i < size2; i ++)

{

id\_list = id\_list + ids[i] + ", ";

}

id\_list = id\_list.substr(0,id\_list.length() - 2);

if (size2 < size1)

{

return 0;

}

else

{

return 1;

}

}

//对已定义的函数名、参数名、const、type和变量名进行查重，有重复则删除id

//返回值为0表示函数名有重复，返回值为1函数名表示无重复

int check\_repeat\_func(string id)

{

int result = 1;

if (get\_typedef\_value(id) != "not exist" || get\_const\_value(id) != "not exist")

{

result = 0;

}

struct symbol\* search\_var;

search\_var = func\_head\_node->symbol\_list->next\_symbol;

while (search\_var != NULL)

{

if (search\_var->name == id)

{

result = 0;

}

search\_var = search\_var->next\_symbol;

}

struct function\* search\_func;

search\_func = func\_head\_node;

while (search\_func != NULL)

{

if (search\_func->name == id)

{

result = 0;

}

search\_func = search\_func->next\_function;

}

return result;

}

//对已定义的函数名、参数名、const、type和变量名进行查重，有重复则删除id

//返回值为0表示有重复，返回值为1表示无重复

int check\_repeat\_definition(string &id\_list)

{

int result = 1;

string new\_list;

vector<string> ids;

if (id\_list != "")

{

new\_list = del\_space(id\_list);

}

else

{

new\_list = "";

}

ids = split(new\_list,",");

//是否与当前函数名重复

for (int i = 0; i < ids.size(); i ++)

{

vector<string>::iterator it;

if (ids[i] == func\_cur\_node->name)

{

it = ids.begin() + i;

ids.erase(it);

result = 0;

}

}

//是否与已定义参数名重复

struct symbol\* search\_sym = func\_cur\_node->symbol\_list;

struct symbol\* search\_para = NULL;

if (search\_sym != NULL)

{

search\_para = search\_sym->parameters;

}

if (search\_para != NULL)

{

while (search\_para->name != "")

{

for (int i = 0; i < ids.size(); i ++)

{

if (ids[i] == search\_para->name)

{

vector<string>::iterator it;

it = ids.begin() + i;

ids.erase(it);

result = 0;

}

}

search\_para = search\_para->next\_symbol;

}

}

for (int i = 0; i < ids.size(); i ++)

{

vector<string>::iterator it;

//是否与已定义const重复

if (get\_const\_value(ids[i]) != "not exist")

{

it = ids.begin() + i;

ids.erase(it);

result = 0;

}

if (ids.size() == 0)

{

break;

}

//是否与已定义type重复

if (get\_typedef\_value(ids[i]) != "not exist")

{

it = ids.begin() + i;

ids.erase(it);

result = 0;

}

}

//是否与已定义变量名重复

if (search\_sym != NULL)

{

search\_sym = func\_cur\_node->symbol\_list->next\_symbol;

}

while (search\_sym != NULL)

{

for (int i = 0; i < ids.size(); i ++)

{

if (ids[i] == search\_sym->name)

{

vector<string>::iterator it;

it = ids.begin() + i;

ids.erase(it);

result = 0;

}

}

search\_sym = search\_sym->next\_symbol;

}

id\_list = "";

for (int i = 0; i < ids.size(); i ++)

{

id\_list = id\_list + ids[i] + ", ";

}

id\_list = id\_list.substr(0,id\_list.length() - 2);

return result;

}

//id\_list自身内部查重与已定义变量,const,type的查重

int check\_var\_declaration(string &id\_list,string &type)

{

int ret = 1;

if (in\_record == 1)

{

if (check\_in\_ids(id\_list) == 0)

{

ret = 0;

}

}

else

{

if (check\_in\_ids(id\_list) == 0 || check\_repeat\_definition(id\_list) == 0)

{

ret = 0;

}

if (id\_list != "")

{

add\_symbol(id\_list, type);

}

}

return ret;

}

//将const类型加入const\_table中

void add\_const(string id, string replace, string replace\_type)

{

struct const\_node new\_const;

bool is\_id = false;

new\_const.name = id;

new\_const.const\_type = replace\_type;

new\_const.value = replace;

const\_table.push\_back(new\_const);

func\_cur\_node->const\_num ++;

const\_cur\_pos ++;

}

//将type类型加入type\_table中

void add\_typedef(string id, string type)

{

struct type\_node new\_type;

new\_type.name = id;

new\_type.type = type;

type\_table.push\_back(new\_type);

func\_cur\_node->type\_num ++;

type\_cur\_pos ++;

}

//翻译参数定义和变量定义

string definition\_trans(string para, bool is\_para)

{

bool is\_var = false;

//传引用的参数与传值参数翻译不同

if (para.find("var+") != -1)

{

is\_var = true;

para = para.substr(4);

}

string result;

string id\_list;

vector<string> ids;

if (para.find("[") == -1 && para.find("{") == -1)

//基本类型的翻译

{

int pos = para.find(":");

id\_list = para.substr(pos + 1);

ids = split(id\_list, ",");

string type = para.substr(0, pos);

if (is\_para)

{

for (int i = 0; i < ids.size(); i++)

{

if (is\_var)

{

result = result + type + " &" + ids[i] + ", ";

}

else

{

result = result + type + " " + ids[i] + ", ";

}

}

result = result.substr(0, result.length() - 2);

}

else

{

result = type + " " + id\_list;

}

}

if (para.find("struct") == 0)

//record类型的翻译

{

int pos = para.find("}");

string type = para.substr(0, pos + 1);

id\_list = para.substr(pos + 2);

ids = split(id\_list, ",");

if (is\_para)

{

for (int i = 0; i < ids.size(); i++)

{

if (is\_var)

{

result = result + type + " &" + ids[i] + ", ";

}

else

{

result = result + type + " " + ids[i] + ", ";

}

}

result = result.substr(0, result.length() - 2);

}

else

{

result = type + " " + id\_list;

}

}

else if (para.find("[") != -1)

//数组类型的翻译

{

para = para.substr(5);

int pos = para.find(":");

int pos1 = para.find("[");

int pos2 = para.find("]");

id\_list = para.substr(pos + 1);

ids = split(id\_list, ",");

string type = para.substr(0, pos1);

string dimension = para.substr(pos1 + 1, pos2 - pos1 - 1);

vector<string> dim\_list = split(dimension, ",");

string final\_dim;

for (int j = 0; j < dim\_list.size(); j++)

{

int pos3 = dim\_list[j].find("|");

string low = dim\_list[j].substr(0, pos3);

string high = dim\_list[j].substr(pos3 + 1);

int range, low\_bound, high\_bound;

low\_bound = atoi(low.c\_str());

high\_bound = atoi(high.c\_str());

range = high\_bound - low\_bound + 1;

stringstream ss;

ss << range;

string s = ss.str();

final\_dim = final\_dim + "[" + s + "]";

}

if (is\_para)

{

for (int i = 0; i < ids.size(); i++)

{

if (is\_var)

{

result = result + type + " &" + ids[i];

}

else

{

result = result + type + " " + ids[i];

}

result = result + final\_dim + ", ";

}

result = result.substr(0, result.length() - 2);

}

else

{

result = result + type + " ";

for (int i = 0; i < ids.size(); i ++)

{

result = result + ids[i] + final\_dim + ", ";

}

result = result.substr(0, result.length() - 2);

}

}

return result;

}

int main() //程序主函数，这个函数也可以放到其它.c, .cpp文件里

{

FILE\* fp1 = fopen(sFile, "r");

if (fp1 == NULL)

{

printf("cannot open %s\n", sFile);

return -1;

}

extern FILE\* yyin,yyout; //yyin和yyout都是FILE\*类型

yyin = fp1; //yacc会从yyin读取输入，yyin默认是标准输入，这里改为磁盘文件。yacc默认向yyout输出，可修改yyout改变输出目的

printf("-----begin parsing %s\n", sFile);

yyparse(); //使yacc开始读取输入和解析，它会调用lex的yylex()读取记号

puts("-----end parsing");

fclose(fp1);

return 0;

}

# 8.测试报告

## 8.1测试环境

Linux Ubuntu 16.04

## 8.2测试计划及结果

**测试报告**

**测试环境**：Linux Ubuntu 16.04

**测试用例**：

### 一、程序框架

1.缺少end后面的.

program test(input, output);

begin

end

预期结果：

#include<stdio.h>

int main()

{

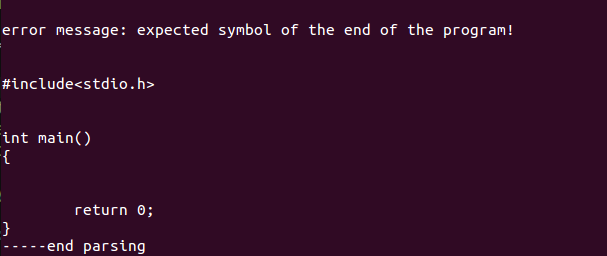
return 0;

}

并报错。

测试结果与分析：

在PASCAL程序最后少了“.”符号，报错。



2.正确

program test(input, output);

var x, y:array [1..100, -3..+50] of integer; z: real;

function f1( var s : real; t: integer): real;

var b: record x:integer; y:real end;

begin

s := x[4,10];

y[60, 40] := t

end;

begin

write(1)

end.

预期结果：

#include<stdio.h>

int x[100][54], y[100][54];

float z;

float f(float &s, int t)

{

struct

{

int x;

float y;

}b;

s = x[3][13];

y[59][43] = t;

}

int main()

{

printf(“%d”,1);

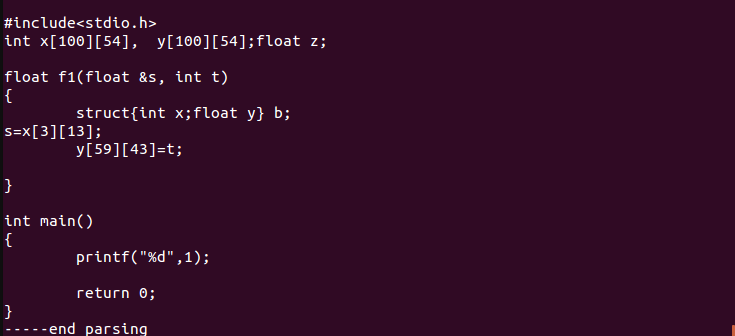
return 0;

}

不报错。

测试结果与分析：

翻译程序正确！



3.语句间缺少分号

program test(input, output);

const a = 1

type q = integer

var ary : array [1..100] of integer; x : integer;

function f1(var a, b : integer) : char;

begin

ary[100] := ary[100] + 1

x := 2

end;

begin

x := x + a

end.

预期结果：

#include<stdio.h>

const int a = 1;

typedef int q;

int ary[100];

int x;

char f1(int &a, int &b)

{

ary[99] = ary[99] + 1;

x = 2;

}

int main()

{

x = x + a;

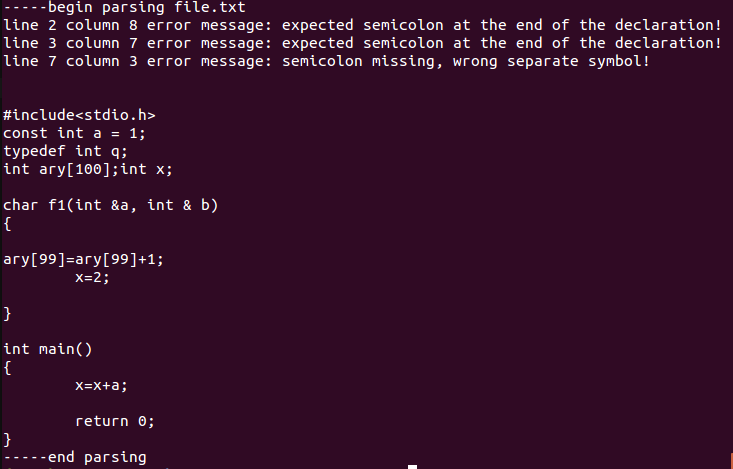
return 0;

}

并报错。

测试结果与分析：

const和type定义的语句结尾缺少分号，ary数组赋值语句的结尾也缺少分号，编译后经过了错误处理使编译过程继续完成并最终报错。



4.多余分号

program test(input, output);

var ary : array [1..100] of integer;

x : integer;

function f1(var a, b : integer;c,d:char;) : char;

begin

ary[100] := ary[100] + 1;

ary[99] := 2;

end;

begin

x := x + 1;

x := 1\*8 - 4;

end.

预期结果：

#include<stdio.h>

int ary[100];

int x;

char f1(int &a, int &b, char c, char d)

{

ary[99] = ary[99] + 1;

ary[98] = 2;

}

int main()

{

x = x + 1;

x = 1\*8 – 4;

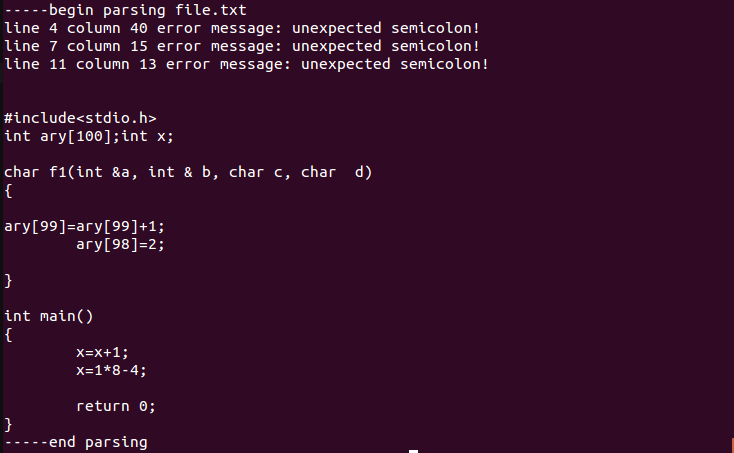
return 0;

}

并报错。

测试结果与分析：

PASCAL程序f1函数的参数最后多加了; ，以及f1函数和主函数最后一条赋值语句最后都多加了分号，故报错。



### 二、常量声明部分

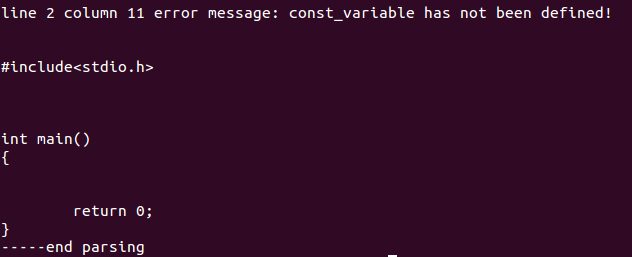
1.声明错误

program test(input, output);

const x = ‘a;

begin

end.



2.科学计数法表达错误

program test(input, output);

const x = 1.E+6;

begin

end.

预期结果：词法错误，直接终止编译，报错。

测试结果与分析：常数x科学计数法的小数点后应该为数字，故发生词法错误，报错。



3.常量赋值错误（赋值号错误，赋值常量不存在）

program test(input, output);

const x := 1;y = a;

begin

end.

预期结果：

#include<stdio.h>

const int x = 1;

int main()

{

return 0;

}

并报错。

测试结果与分析：

const的赋值使用了:=符号而不是=，使用变量a给常量y赋值时，由于常量y之前未定义故出现错误，编译程序将两处错误均报出，并不翻译出错的定义。



4.常量重定义

program test(input, output);

const x = 1;x = 2;

function f1(a:integer):integer;

const a = 0;f1 = 3.2;

begin

end;

begin

end.

预期结果：

#include<stdio.h>

const int x = 1;

int f1(int a)

{

}

int main()

{

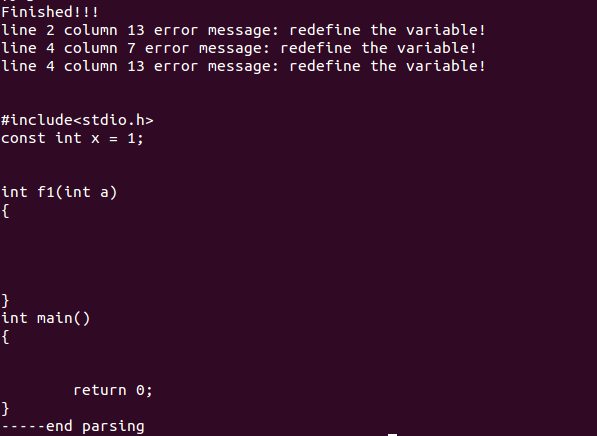
return 0;

}

并报错。

测试结果与分析：

主函数const定义x变量与之前已定义的变量重复，报错，并不翻译重定义语句。f1函数中常量a的定义与参数中a的定义重复，常量f1的定义与函数名f1重复，都进行报错。



5.常量改变

program test(input, output);

const x = 1;

begin

x := 2;

end.

预期结果：

#include<stdio.h>

const int x = 1;

int main()

{

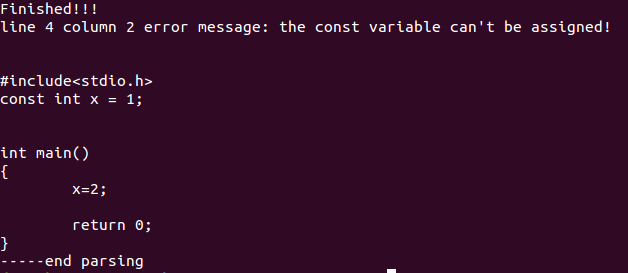
x = 2;

}

并报错。

测试结果与分析：

常量定义后值不可改变，报错。



6.正确

program test(input, output);

const x = 1;y = 2;

function f1(a:integer):integer;

const h = 0;i = h;

begin

end;

begin

end.

预期结果：

#include<stdio.h>

const int x = 1;

const float y = 2.3;

int f1(int a)

{

const int h = 0;

const int i = h;

}

int main()

{

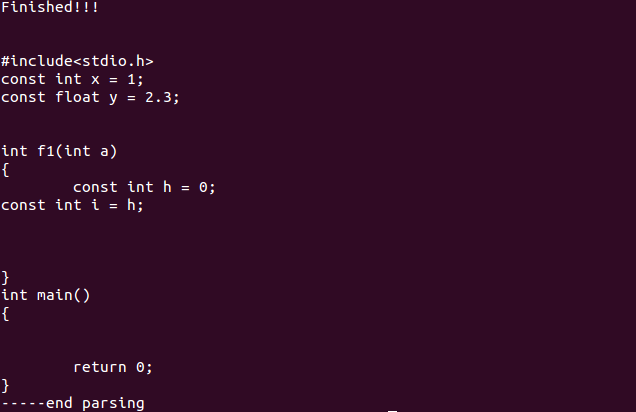
return 0;

}

不报错。

测试结果与分析：

程序正确无误。



### 三、变量声明部分

1.变量标识符错误

program test(input, output);

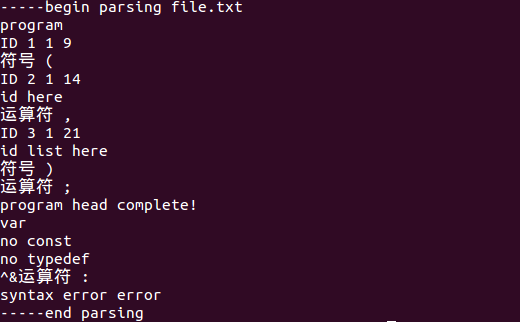
var ^& : integer;

begin

end.

预期结果：标识符词法错误，编译停止，报错。

测试结果与分析：标识符id必须以字母开头，且不含除字母和数字之外的字符。直接报错。



2.定义中的符号错误

program test(input, output);

var a,. b: integer; c. , d: char;

begin

end.

预期结果：

#include<stdio.h>

int a, b;

char c, d;

int main()

{

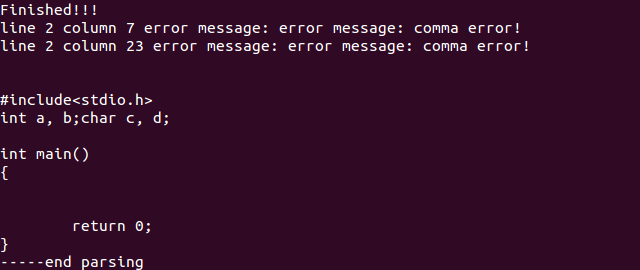
return 0;

}

并报错。

测试结果与分析：

第二行id\_list中出现了错误的字符，编译程序跳过错误继续编译，最终输出报错。



3.重复定义

program test(input, output);

const a = 1;

type a = integer;

var a, b: integer; a : char;

begin

end.

预期结果：

#include<stdio.h>

const int a = 1;

int b;

int main()

{

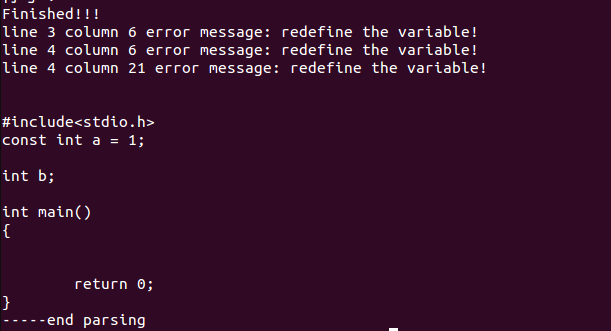
return 0;

}

并报错。

测试结果与分析：

type类型定义与之前已定义的const常量重复，报错；var 定义中的变量a也与已经定义的变量重复，报错；不翻译重复定义的语句。



program test(input,output);

const a = 0;b = 1;

type c = integer;

var d,a,c:integer;

h:c;

b:array[-1..+100] of integer;

k:record a:integer end;

procedure f1(b:integer;w:char):integer;

const f1 = 2;m = g;

type l = integer;p = q;

var b:integer;f1:char;v:integer;

begin

read(v,b[1]);

write(v,1,'a',1+1.2);

end;

begin

end.

预期结果：

#include<stdio.h>

const int a = 0;

const int b = 1;

typedef int c;

int d;

int h;

struct

{

int a;

}k;

void f1(int b, char w)

{

typedef int l;

int v;

scanf(“%d%d”,&v,&b[i]);

printf(“%d%d%c%f”,v,1,’a’,1+1.2);

}

int main()

{

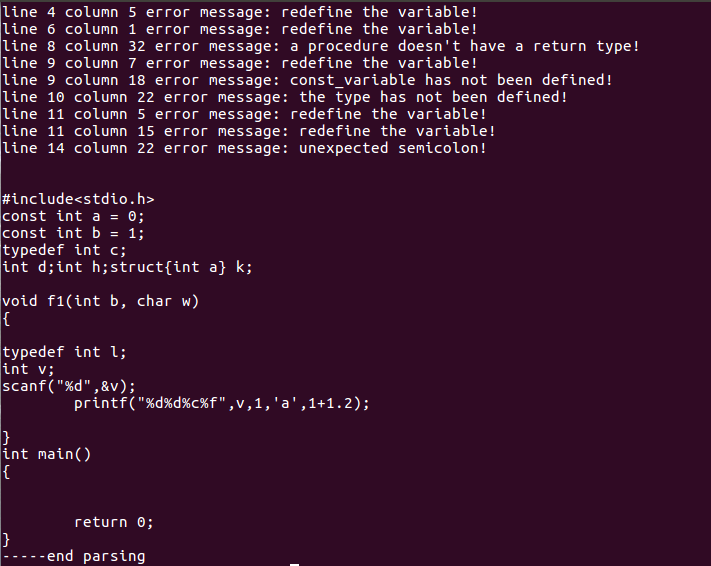
return 0;

}

并报错。

测试结果与分析：

主函数var定义与之前const和type定义重复，报错；过程f1中const定义与函数名重复，并由未定义的常量值赋值，报错；type定义中也存在未定义的type类型赋值，报错；var变量定义与参数和函数名重复，报错。出现错误的定义语句均不翻译。



4.数组内部定义错误

program test(input, output);

var a: array [1...100] of integer;

begin

end.

预期结果：

#include<stdio.h>

int a[100];

int main()

{

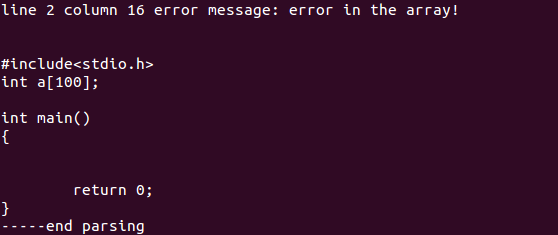
return 0;

}

并报错。

测试结果与分析：

数组定义时多了一个点，报错；



5.正确

program test(input, output);

var a, b: integer;

function f1(a:integer):integer;

var c:char;d:array [1..100] of integer;

begin

end.

预期结果：

#include<stdio.h>

int a, b;

int f1(int a)

{

char c;

int d[100];

}

int main()

{

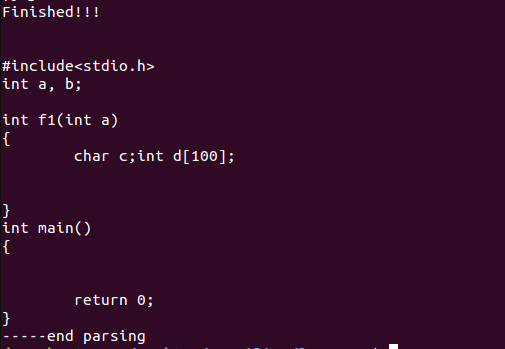
return 0;

}

无报错。

测试结果与分析：

程序无错误，输出翻译结果。



### 四、过程和函数声明部分

1.()中无参数

program test();

const x = 1;

begin

end.

预期结果：

#include<stdio.h>

const int x = 1;

int main()

{

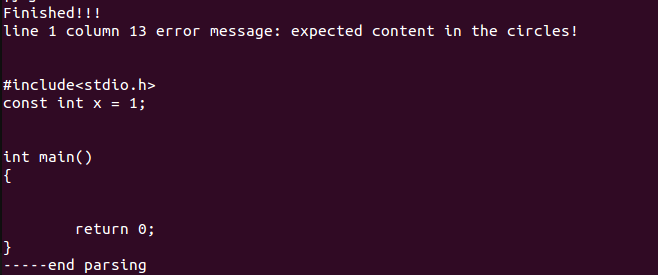
return 0;

}

报错。

测试结果与分析：

program后的()中必须要有参数，括号中为空则报错。



2.缺少()

program test;

const x = 1;

begin

end.

预期结果：

#include<stdio.h>

const int x = 1;

int main()

{

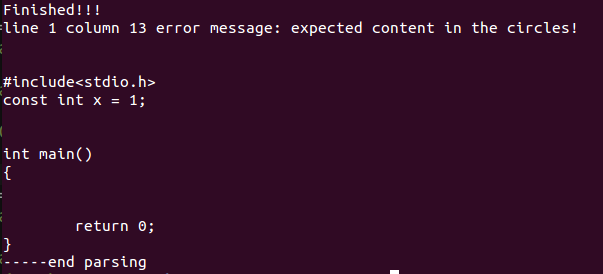
return 0;

}

报错。

测试结果与分析：

program后的必须带有括号和参数，没有则继续编译并报错。



3.procedure有返回值

program test(input, output);

procedure p1 (var a , b: integer) : integer;

begin

end;

begin

end.

预期结果：

#include<stdio.h>

void p1(int &a, int &b)

{

}

int main()

{

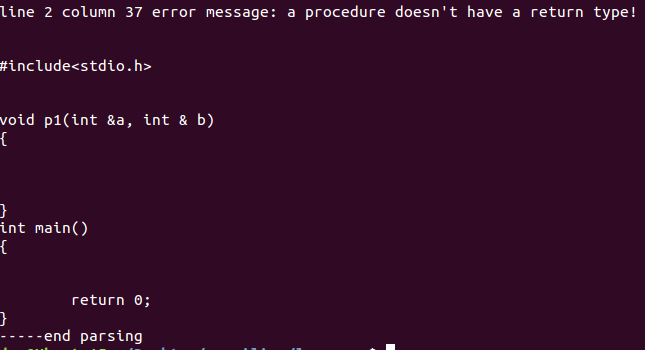
return 0;

}

报错。

测试结果与分析：

PASCAL的procedure不存在返回值，若有返回值则报错，但是继续编译下去。



4.function没有返回值

program test(input, output);

function f1 (var a , b: integer) ;

begin

end;

begin

end.

预期结果：

#include<stdio.h>

int f1(int &a, int &b)

{

}

int main()

{

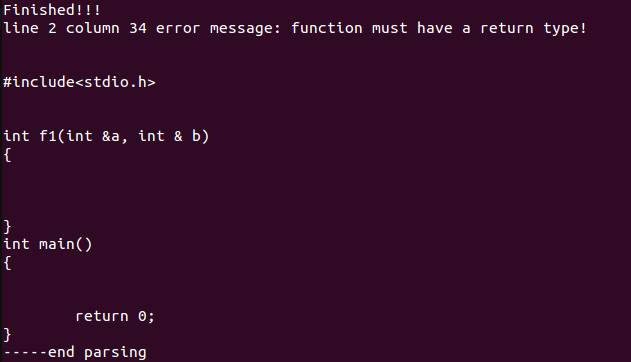
return 0;

}

报错。

测试结果与分析：

PASCAL中函数需要有返回值，如果不存在返回值则报错并继续编译下去。



5.定义缺少类型

program test(input, output);

procedure p1 (var a , b);

begin

end;

begin

end.

预期结果：

#include<stdio.h>

void p1(int &a, int &b)

{

}

int main()

{

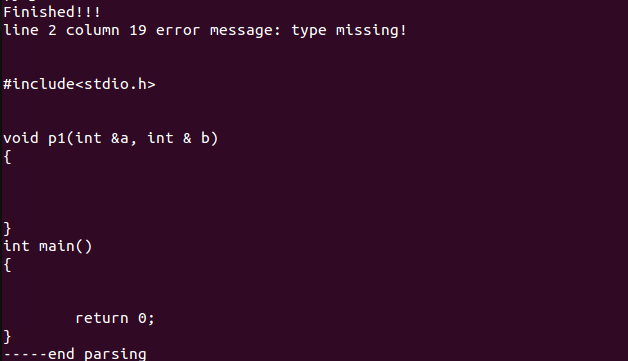
return 0;

}

并报错。

测试结果与分析：

函数或者过程中的参数没有类型，编译器自动为赋予int类型，并报错。



6.变量无定义

program test(input, output);

procedure p1 (var a , b: integer);

begin

end;

procedure p2( var c, d: char);

begin

a := 2

end;

begin

end.

预期结果：

#include<stdio.h>

void p1(int &a, int &b)

{

}

void p2(char &c, char &d)

{

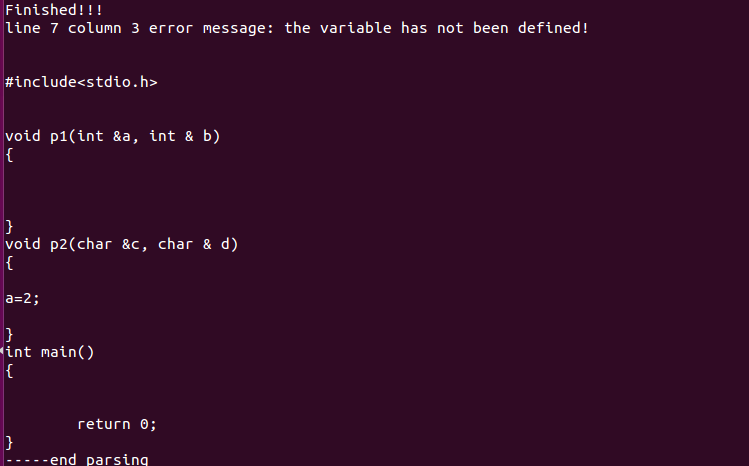
a = 2;

}

报错。

测试结果与分析：

变量a仅为过程p1的参数，在过程p2中使用则会报变量未定义的错误。



### 五、函数和过程调用部分

1.类型赋值错误

program test(input, output);

var x: boolean;

function f1(var a, b : integer) : char;

begin

end;

begin

x := f1(1, 2)

end.

预期结果：

#include<stdio.h>

bool x;

char f1(int &a, int &b)

{

}

int main()

{

x = f1(1,2);

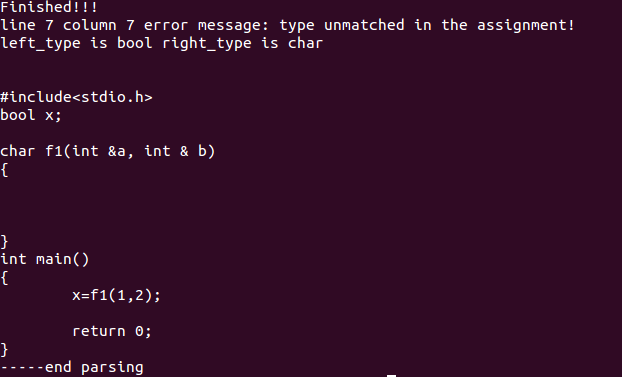
return 0;

}

报错。

测试结果分析：

x变量为bool类型，而函数f1的返回值为char类型，赋值语句两端类型不一致，报错。



2.参数类型错误

program test(input, output);

var x: boolean;ary : array [1..100] of integer;

function f1(var a, b : integer) : char;

begin

end;

begin

x := f1(1, ary);

end.

预期结果：

#include<stdio.h>

bool x;

int ary[100];

char f1(int &a, int &b)

{

}

int main()

{

x = f1(1,ary);

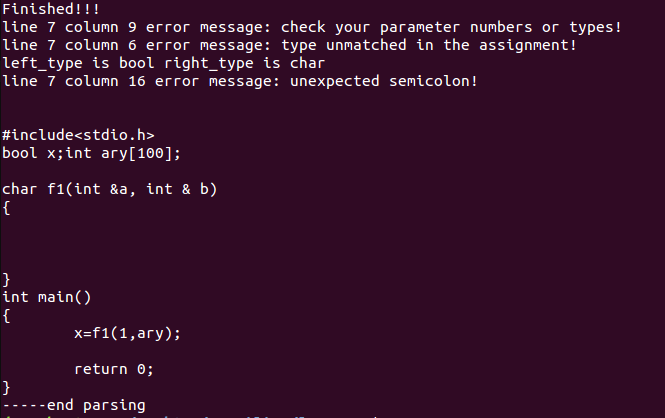
return 0;

}

报错

测试结果分析：

函数f1的两个形参类型为integer类型，但是实参ary仅仅为数组名，参数类型错误。



3.参数个数错误

program test(input, output);

var x: boolean;

function f1(var a, b : integer) : char;

begin

end;

begin

x := f1(1, 2, 3)

end.

预期结果：

#include<stdio.h>

bool x;

int ary[100];

char f1(int &a, int &b)

{

}

int main()

{

x = f1(1,2,3);

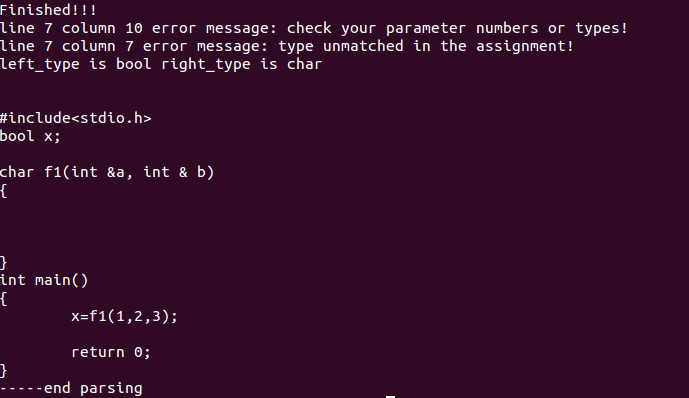
return 0;

}

报错。

测试分析结果：

实参数目有三个，而形参数目只有两个，参数个数不匹配，报错。



### 六、程序体部分

1.赋值语句两边类型不匹配

program test(input, output);

var x, y:array [1..100, -3..+50] of integer; z: real;

function f1( var s : real; t: integer): real;

var b: record x:integer; y:real end;

begin

s := x[4,10];

y[60, 40] := t;

b.x := (t + 2)/5;

t := b.x \* b.y;

b.x := b.y

end;

begin

write(1);

end.

预期结果：

#include<stdio.h>

int x[100][54], y[100][54];

float z;

float f1(float &s, int t)

{

struct

{

int x;

float y;

}b;

s = x[3][13];

y[59][43] = t;

b = ( t + 2 ) / 5;

t = b.x\*b.y;

b.x = b.y;

}

int main()

{

printf(“%d”,1);

return 0;

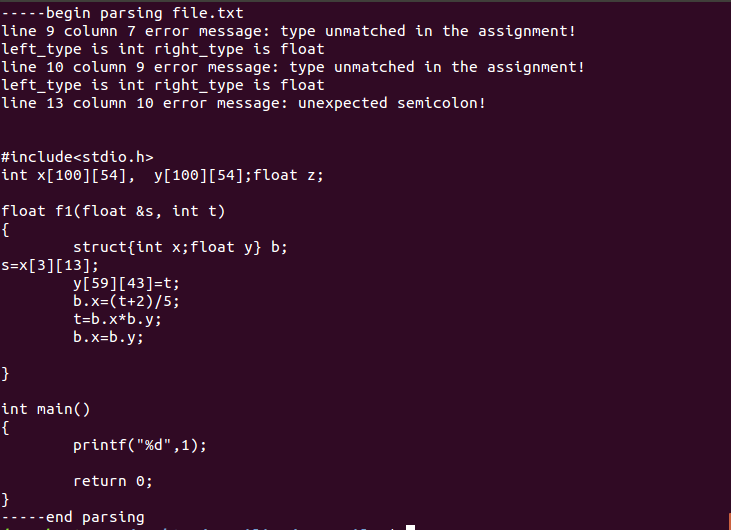
}

报错。

测试结果分析：

变量t为integer型，b.x乘b.y为real型，两边类型不一致，报错。

b.x为integer型，b.y为real型，同样报错。



2.表达式缺少运算对象

program test(input, output);

var x: integer;

begin

x := 1 +;

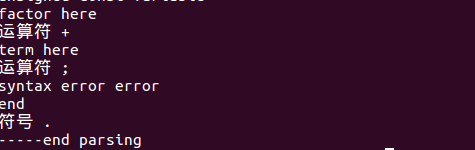
end.

预期结果：

表达式缺少运算对象，词法错误，编译终止，报错。

测试结果与分析：

识别到错误的单词，报错。



3.赋值号错误

program test(input, output);

const y = 1;

var x: integer;

begin

x = y;

x := y + 1

end.

预期结果：

#include<stdio.h>

const int y = 1;

int x;

int main()

{

x = y;

x = y + 1;

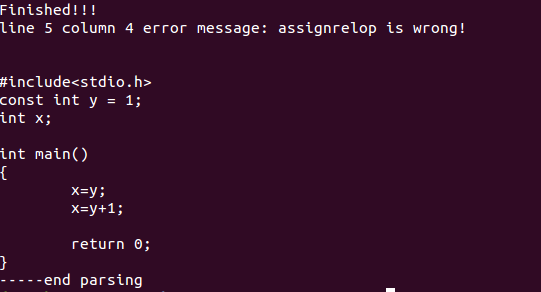
return 0;

}

报错。

测试结果分析：

将赋值号误用为等号，报错。



4.for语句使用错误,少了do

program test(input, output);

const y = 1;

var x: integer;

begin

for x := 0 to 9

x := y

end.

预期结果：

#include<stdio.h>

const int y = 1;

int x;

int main()

{

for (x = 0; x <= 9; x ++)

{

x = y;

}

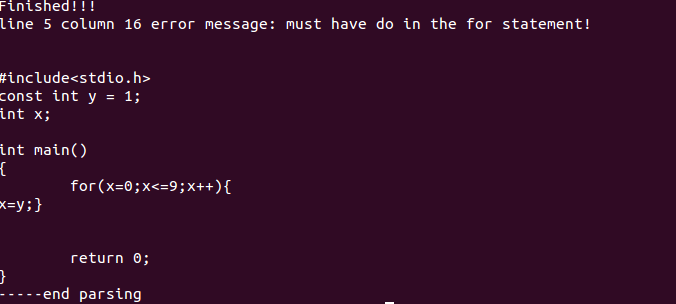
return 0;

}

报错。

测试结果与分析：

for语句中缺少do关键字，报错。



5. case语句缺少end

program test(input, output);

const y = 1;

var x: integer;

begin

case x of

1, 2: write(1);

3, 4: write(2)

x := y + 1;

x := x + x\*5

end.

预期结果：

#include<stdio.h>

const int y = 1;

int x;

int main()

{

switch(x)

{

case 1:

printf(“%d”,1);

break;

case 2:

printf(“%d”,1);

break;

case 3:

printf(“%d”,2);

break;

case 4:

printf(“%d”,2);

break;

}

x = y + 1;

x = x + x \* 5;

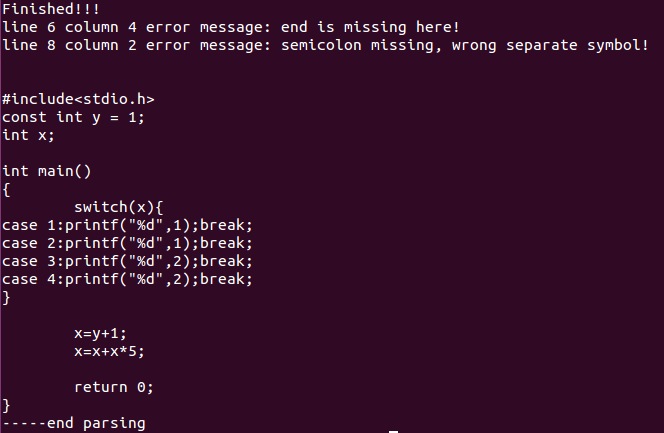
return 0;

}

报错。

测试结果分析：

case语句最后缺少了关键字end以及分隔符分号，报错。



6.case中重复

program test(input, output);

const y = 1;

var x: integer;

begin

case x of

1, 2: write(1);

2, 3: write(2);

4, 4: write(3)

end

end.

预期结果：

#include<stdio.h>

const int y = 1;

int x;

int main()

{

switch(x)

{

case 1:

printf(“%d”,1);

break;

case 2:

printf(“%d”,1);

break;

case 3:

printf(“%d”,2);

break;

case 4:

printf(“%d”,3);

break;

}

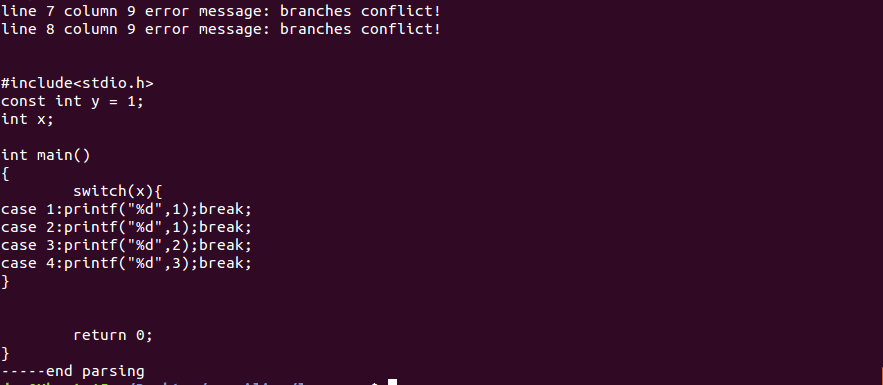
return 0;

}

报错。

测试结果分析：

case分支中的常数重复，则保留最开始的case分支，删除重复分支，报错。



7.除数为0

program test(input, output);

const y = 1;

var x: integer;

begin

x := x/0;

x := y/0;

end.

预期结果：

#include<stdio.h>

const int y = 1;

int x;

int main()

{

x = x / 0;

x = y / 0;

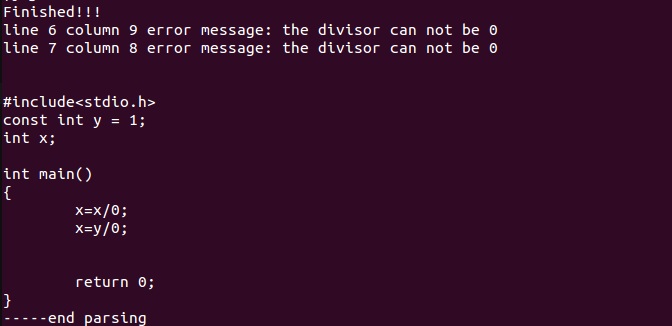
return 0;

}

报错。

测试结果分析：

除数部分直接为0，报错。



七、其他错误

1. 数组下标越界，数组维数错误

program test(input, output);

var ary : array [1..100] of integer;

function f1(var a, b : integer) : char;

begin

ary[103] := ary[117] + 1;

ary[1,2] := a + 1

end;

begin

x = f1(1, ary)

end.

预期结果：

#include<stdio.h>

int ary[100];

char f1(int &a, int &b)

{

ary[102] = ary[116] + 1;

ary[1][2] = a + 1;

}

int main()

{

x = f1(1,ary);

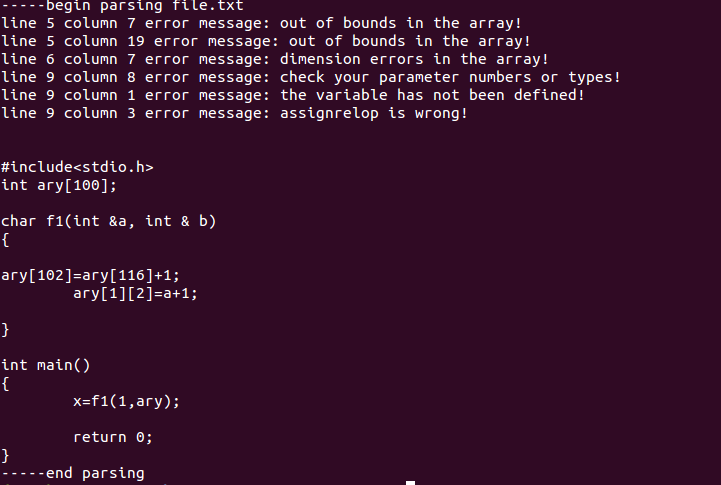
return 0;

}

并报错。

测试结果分析：

ary数组使用时下标越界且数组维度不匹配，报错。



2. 类型使用错误，非数组变量当成数组

program test(input, output);

var ary : array [1..100] of integer;

x : integer;

function f1(var a, b : integer) : char;

begin

ary[100] := ary[100] + 1;

end;

begin

x := f1(1, 2);

x[1] := x[1] + 1 ;

end.

预期结果：

#include<stdio.h>

int ary[100];

int x;

char f1(int &a, int &b)

{

ary[99] = ary[99] + 1;

}

int main()

{

x =f1(1,2);

x = x + 1;

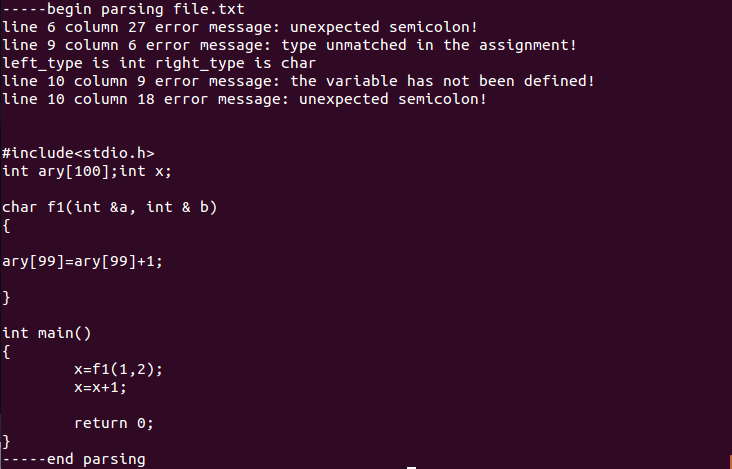
return 0;

}

并报错。

测试结果分析：

x变量仅仅是一个整型变量，却在调用时当做数组调用，报错。



3. 括号不匹配

program test(input, output;

var ary : array [1..100] of integer;

function f1 (var a, b : integer) : char;

begin

ary[101] := ary[101] + 1;

end;

begin

x := f1(1, ary);

end.

预期结果：

#include<stdio.h>

int ary[100];

char f1(int &a, int &b)

{

ary[100] = ary[100] + 1;

}

int main()

{

x =f1(1,2);

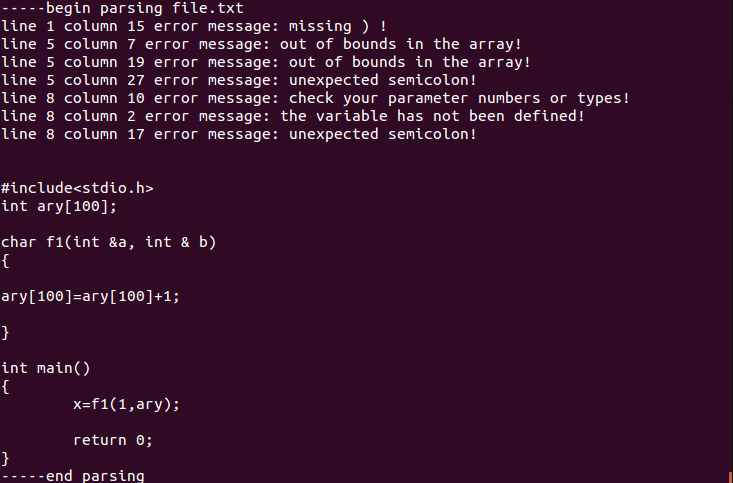
return 0;

}

并报错。

测试结果分析：

program声明时右边括号缺失，报错。



4. if中不是布尔表达式

program test(input, output);

var ary : array [1..100] of integer;

x : integer;

function f1(var a, b : integer;c,d:char) : char;

begin

ary[100] := ary[100] + 1;

ary[99] := 2

end;

begin

x := f1(1, 2);

if x+2 then

x[1] := x[1] + 1

else

x[1] := (x[1]+1)mod(2\*(3+x[1]))

end.

预期结果：

#include<stdio.h>

int ary[100];

int x;

char f1(int &a, int &b, char c, char d)

{

ary[99] = ary[99] + 1;

ary[98] = 2;

}

int main()

{

x = f1(1,2);

if (x + 2)

{

x = x + 1;

}

else

{

x = (x+1)%(2\*3+x));

}

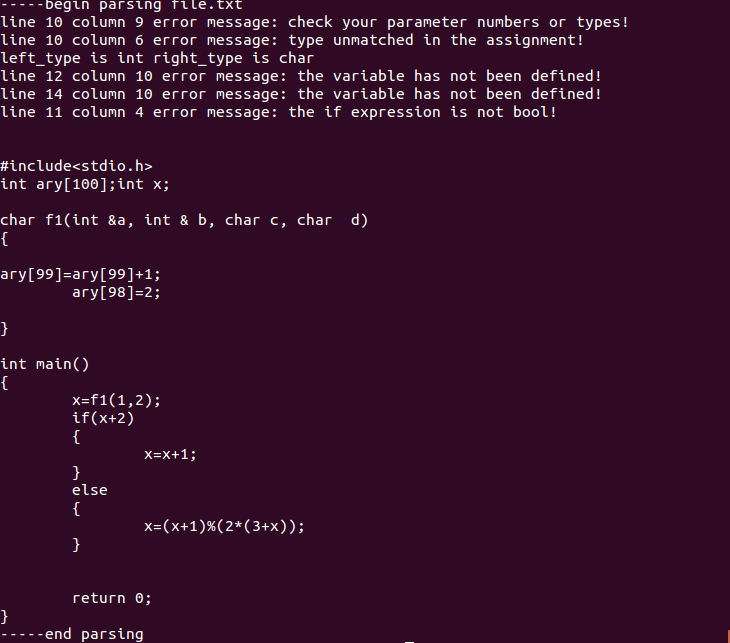
return 0;

}

并报错。

测试结果分析：

if中判断语句不为bool型，报错。



# 9.实验总结

## 9.1主要问题

### 9.1.1 Flex和Yacc的连接问题

在研究Flex和Yacc连接机制的时候，一直出现变量重定义的问题，后来理解了二者之间数据传递的机制后解决。Lex中yylex()函数保存识别到的字符信息，并返回一个记号，Yacc在收到一个记号后就通过文法产生式进行相应的动作。

### 9.1.2文法修改

在编写Yacc的时候，我们小组尝试在文法中间嵌入语义动作，实现在移进过程中的翻译，但是发现结果并不尽如人意，原因是Bison中只能够在文法最后规约的时候执行相应的语义动作，为了实现符号表操作，我们因此将部分文法修改，引入了新的非终结符，在其规约时完成原来嵌入在文法中的动作。如在识别到函数名时，创建一个新的函数表节点，然后当函数的返回值类型被识别到后，在回填函数表。

### 9.1.3文法扩充

我们小组在原来文法的基础上加入了read、write和writeln语句的分析和翻译。另外为了对错误的提示和容错，新增了识别错误语句的文法。

### 9.1.4错误处理

设计识别错误的文法时，遇到了增加新的文法后出现规约-规约冲突的情况。

### 9.1.5符号表操作

由于我们的符号表数据结构是以指针链表构建的，在相关操作时不可避免的会发生各种指针错误，这个过程也是一个非常头疼的过程，因为Bison的调试信息output文件过于复杂，涉及很多状态转移的信息，我们在调试时只能通过在语句之间增加输出信息这种比较费力的方式寻找错误点，十分繁重。

## 9.2改进建议

### 9.2.1符号表设计

目前我们设计的函数表和符号表结构都比较复杂，维度比较高，可以考虑优化成多表结构，并且在每个表中加入索引域，这样查询涉及的指针操作会减少很多，代价就是需要多张表的管理。

### 9.2.2记录类型的嵌套

我们目前的翻译和分析没有处理记录类型中套入复杂类型的情况，只是简单变量，这和我们目前的识别机制有关，后面可以进行改进。

### 9.2.3词法分析错误处理

我们目前的词法分析遇到无法分析的字符时会把错误传递到语法分析中，本身不具备检错和纠错的能力。可以后期进行改进。

### 9.2.4动态检查

目前的错误检查和类型检查都只涉及静态检查，例如数组越界的检查只涉及立即数的情况，变量或者表达式作为下标索引的情况没有进行相应的错误检查。

### 9.2.5 Yacc错误处理机制

Yacc错误处理机制是通过一个error的终结符去比配任意字符，直到遇到后面的一个有效字符为止，这种机制如果想要发现一个源文件中多有的错误，需要将其嵌入到很多文法产生式中的不同地方，我们因此只选择了一部分错误。其他诸如类型不匹配的错误考虑较为全面。

## 9.3心得体会

耿林桐：

在本次为期八周的编译原理小学期中，在前期设计方面，我主要参与了Pascal-c语言编译器的整体设计方面，大家大概一起讨论了4次，最后确定了整体的结构流程-采用lex yacc工具以一次编译的方式进行编译器的词法分析语法分析语义分析目标代码生成。并且参与了对各个错误情况，动态编译方面进行了方案的确定。

在后期代码编写阶段，我个人完成了lex词法分析的编写，并且对于一些注释，个别词的识别错误进行了识别。并且协助负责语法分析的同学进行语法分析的部分工作。对于语法分析方面一些空指针，链表创建错误，多次输出等错误进行了程序的debug。

对于这次整个设计编写过程，我主要有以下三点感受:

第一点:熟悉了编译性语言在链接编译过程中，它的编译器是如何工作的。让我对于一些编译型语言，比如c语言，Pascal等从编译的角度有了更加深入的了解。

第二点:对于Linux下lex yacc等工具有了进一步的认识和使用经验，并且对于Linux中make makefile等有了一些概念，对c++debug有了更加深刻的认识。

第三点:对于团队合作有了更加清楚的理解和感受，我们在八周实验的整个过程中，有很清晰的团队分工，并且能够一起合作解决问题，使得发现问题，解决问题的效率大大提升。

综上三点，就是我对于本次编译原理实验的感受，很感谢有这样的机会去做一个相对比较完整的项目，对于个人还是团队都有着很大的收获。

田阔：

通过这次编译原理课程设计，对于编译过程特别是语法语义分析部分有了十分深刻的理解和认识。亲自设计翻译方案，设计符号表的结构，设计函数的过程充满了各种发现问题和解决问题的曲折和挑战，从最初的框架到一步步地丰富，让我对原来的理论知识有了充分的运用，同时自己的编码能力、调试能力以及耐心、团队交流沟通能力也得到了锻炼。

项子豪：

这次的编译原理大作业收获很多，看似是上学期的几次小作业结合在一起，其实并不然。不论是使用的文法，还是采用的编译方式都截然不同。在符号表的建立过程中，使用链表将函数表与参数表和符号表连在一起，编译的过程中添加了很多麻烦，是前期设计时没有想到的。在翻译文法时，将subprogram\_declarations和compound\_statement这两个包含的所有文法产生式画在纸上，每个产生式相连接，防止在翻译目标代码的时候找不到方向。

在这八周的实验过程中，将上学期学到的知又深刻学习理解了一遍，感谢有机会和大家一起进行团队合作，遇到不懂的地方，谢谢有人能够帮助我理解并解决问题，受益匪浅。

周尧棋：

此次编译原理大作业我负责的部分是语法、语义分析以及代码生成部分，在实践过程中我不断地加深了对整个编译过程的理解并且提高了自己的代码能力。

首先在学习Lex和Yacc进行编译的过程中就遇到了颇多问题。第一是Lex文件和Yacc文件联合编译的问题，起初没有抓到两个过程的接口以至于编码停滞不前，仔细阅读了Lex和Yacc的使用说明和一些实例后解决了问题；第二是在改进文法时遇到了很多规约-规约冲突，还有在添加嵌套处理是也出现了无法调用的情况，最后在深刻了理解了移进-规约过程后成功地改进了文法，能够使编译顺利的完成。

在设计符号表结构时起初将使用链表将函数表中再包含了符号表和参数表，在编码过程中发现操作十分复杂，链表操作也很容易出错。在以后的编码过程中要学会将各个结构分清，不要混杂在一起使编码变得更为复杂。

最后在错误处理方面，可能是对Lex和Yacc的错误处理机制不够了解，也可能是本身它们的错误处理机制就有局限，导致添加错误处理的过程变得颇为繁琐而且无法考虑到所有的错误情况。这也是我们需要改进的地方。

总的来说，在参与这次编码后，我对于编译的流程和Lex和Yacc的处理机制有了深刻的理解，也对书本上的基础知识有了更深的理解。

林镕智：

通过这次编译原理课程设计，对于语法语义分析部分有了十分深刻的理解和认识。亲自设计翻译方案，思考如何能够改进文法，在错误处理部分，一开始想到的情况很少，在与小组成员交流的过程中，又发现了更多漏洞。这一过程对自身知识的巩固十分有帮助同时自己的编码能力、调试能力以及耐心、团队交流沟通能力也得到了锻炼。

加苏尔·吐尔地：

在这次编译器大作业中，我受益匪浅。自学了不少了知识，还有动手实践了理论，还有团队合作每个人之间的联系，理解和默契。对于上学期在编译原理课上学习的词法分析，语法分析，语义分析，有了更深刻的认识。当然所得到的收获远远不止这些，还有LEX的学习让我有了新知识的储备。团队合作是，大家的互相帮助，有问题一起克服，先分工而运作，之后又拧在一起解决每个人之间，每个任务之间的间隙，也让该工程能够顺利完成。有些欣喜，有些感激。不管在各自学习时的努力，还是各自完成任务时大家都会有困难帮助旁边的人。