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** |
| **张帅帅** | **2016211302** | **2016211149** | **组长** |
| **张津彬** | **2016211302** | **2016211150** |  |
| **张君智** | **2016211301** | **2016211131** |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**计算机学院**

**2019 年 7 月**

目 录

目录

[目 录 2](#_Toc13780219)

[1．课程设计任务和目标 3](#_Toc13780220)

[2．需求分析 3](#_Toc13780221)

[2.1数据流图 3](#_Toc13780222)

[2.2功能说明： 3](#_Toc13780223)

[2.2.1 词法分析功能说明： 3](#_Toc13780224)

[2.2.2语法分析功能说明： 4](#_Toc13780225)

[2.2.3语义分析功能说明： 4](#_Toc13780226)

[2.2.4目标代码生成功能说明 4](#_Toc13780227)

[2.3输入输出数据说明： 4](#_Toc13780228)

[3．总体设计 4](#_Toc13780229)

[3.1 数据结构设计 5](#_Toc13780230)

[3.2总体结构设计 6](#_Toc13780231)

[3.2.1模块划分 6](#_Toc13780232)

[3.2.2模块功能 6](#_Toc13780233)

[3.2.3模块间关系 7](#_Toc13780234)

[3.2.4模块间接口 7](#_Toc13780235)

[4．详细设计 8](#_Toc13780236)

[4.1 词法分析详细设计 8](#_Toc13780237)

[4.1.1 接口描述 8](#_Toc13780238)

[4.1.2 功能描述 8](#_Toc13780239)

[4.1.3 所用数据结构说明 8](#_Toc13780240)

[4.1.4 算法描述 9](#_Toc13780241)

[4.2 语法分析详细设计 9](#_Toc13780242)

[4.2.1 接口描述 9](#_Toc13780243)

[4.2.2 功能描述 9](#_Toc13780244)

[4.2.3 所用数据结构说明 10](#_Toc13780245)

[4.2.4 算法描述 10](#_Toc13780246)

[4.3 符号表详细设计 11](#_Toc13780247)

[4.2.1 接口描述 11](#_Toc13780248)

[4.2.2 功能描述 12](#_Toc13780249)

[4.2.3 所用数据结构说明 12](#_Toc13780250)

[4.3.4 算法描述 12](#_Toc13780251)

[5．源程序清单 12](#_Toc13780252)

[6．程序测试 13](#_Toc13780253)

[6.1 测试环境： 13](#_Toc13780254)

[6.2 测试的功能 13](#_Toc13780255)

[7．课程设计总结 13](#_Toc13780256)

[7.1 体会与收获： 13](#_Toc13780257)

[7.2设计过程中遇到或存在的主要问题及解决方案 14](#_Toc13780258)

[7.3 改进建议 14](#_Toc13780259)

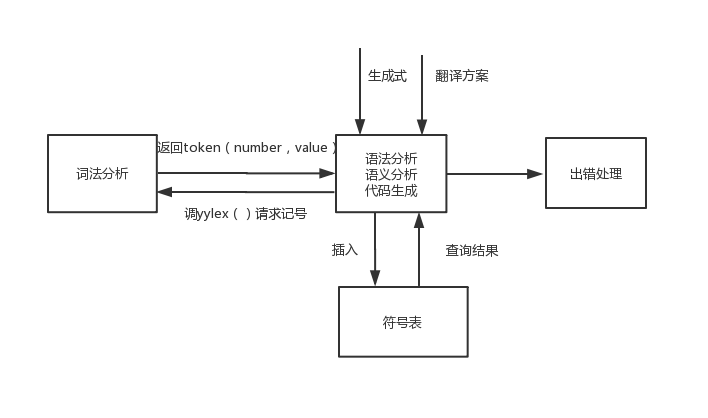
# 1．课程设计任务和目标

设计任务：实现一个Pascal语言 -> C语言的编译器-

设计目标：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语言到C语言的的编译程序。

# 2．需求分析

## 2.1数据流图



## 2.2功能说明：

### 2.2.1 词法分析功能说明：

词法分析主要任务是从左到右逐个字符地对源程序进行扫描，按照Pascal语言的词法规则识别出一个个单词符号，产生用于语法分析的记号序列，并把来自编译程序的错误信息和源程序联系起来，记住单词在源程序中的行/列位置，从而行号可以作为错误信息的一部分提示给用户。

源程序中的关键字（除开头的program和末尾的end之外）前、后必须有空格符或换行符，其它词汇间的空格符是可选的。源程序中的注释：用一对花括号括起来，可以出现在任何单词之后。编译程序应该可以处理注释。

如果词法分析程序从源程序中读入了一个不合法的字符即Pascal语言不包括此字符开头的单词符号。词法分析显示打印错误信息，并跳过这个字符，然后继续识别和分析下一个单词符号。

### 2.2.2语法分析功能说明：

利用Pascal语言的语法规则，对词法分析程序产生的记号流进行规约，检查程序的结构是否正确。并且因为采用语法制导翻译技术，在规约的过程中实现语义分析和代码生成。并对语法分析中出现的错误进行处理。

### 2.2.3语义分析功能说明：

对语句句的意义进⾏行行检查分析，并收集类型等必要信息进行类型查根据规则检查每个运算符及其运算对象是否符合要求；数组的下标是否合法；过程调⽤用时，形参与实参个数、类型是否匹配等

### 2.2.4目标代码生成功能说明

分析Psacal源码，找出其对应的C语言代码，并把适当的语义动作嵌入相应的翻译方案，实现对Pascal到C语言的翻译。

## 2.3输入输出数据说明：

源程序：待分析的Pascal程序代码；

目标代码：生成的目标代码（C语言代码）。

# 3．总体设计

## 3.1 数据结构设计

在该项目的数据结构设计主要体现在符号表的数据结构设计上。

typedef struct \_symbol

{

    const char\* name; // 符号名

    symbol\_type type; // 符号类型

    symbol\_type sub\_type; // 子类型

    unsigned int dimension; // 维数

    int is\_const; // 是否常量

    int is\_reference; // 是否引用

    parameter\_info parameter\_list; // 函数参数

    unsigned int next; // 链域

} symbol;

我们符号表的目标数据结构如上述代码所示。

其中name表示该符号的名称，

Type表示该符号的类型。

Sub\_type表示该符号拥有的子类型

Dimension 表示该符号的维数（多维数组）

Is\_const 表示该符号是否常量

Is\_reference 表示该符号是否被引用

Parameter\_list 函数调用时的参数需要哪些

Next 只想下一个拥有相同散列的符号

## 3.2总体结构设计

### 3.2.1模块划分

根据任务的需要，应该划分为**词法分析、语法分析、语义分析、代码生成、符号表维护**等部分，但我们采用语法制导翻译方案，把语法分析、语义分析、代码生成一起放在语法分析中完成。

### 3.2.2模块功能

**词法分析功能说明：**

词法分析主要任务是从左到右逐个字符地对源程序进行扫描，按照Pascal语言的词法规则识别出一个个单词符号，产生用于语法分析的记号序列，并把来自编译程序的错误信息和源程序联系起来，记住单词在源程序中的行/列位置，从而行号可以作为错误信息的一部分提示给用户。

源程序中的关键字（除开头的program和末尾的end之外）前、后必须有空格符或换行符，其它词汇间的空格符是可选的。源程序中的注释：用一对花括号括起来，可以出现在任何单词之后。编译程序应该可以处理注释。

如果词法分析程序从源程序中读入了一个不合法的字符即Pascal语言不包括此字符开头的单词符号。词法分析显示打印错误信息，并跳过这个字符，然后继续识别和分析下一个单词符号。

**语法分析功能说明：**

利用Pascal语言的语法规则，对词法分析程序产生的记号流进行规约，检查程序的结构是否正确。并且因为采用语法制导翻译技术，在规约的过程中实现语义分析和代码生成。并对语法分析中出现的错误进行处理。

**语义分析功能说明：**

对语句句的意义进⾏行行检查分析，并收集类型等必要信息进行类型查根据规则检查每个运算符及其运算对象是否符合要求；数组的下标是否合法；过程调⽤用时，形参与实参个数、类型是否匹配等

**目标代码生成功能说明**

分析Psacal源码，找出其对应的C语言代码，并把适当的语义动作嵌入相应的翻译方案，实现对Pascal到C语言的翻译。

### 3.2.3模块间关系

语法制导翻译方案以语法分析程序为核心，在进行语法分析的过程中由语法分析程序的分析过程来主导语义分析和中间代码生成。

所以在我们的项目中，由语法分析器调用词法分析器的接口，词法分析器取出源文件中的一个单词并返回；在产生式中加入错误判断，把符号加入符号表中，通过调用符号表的查询接口实现语义的分析。最后在语义动作中加入生成C语言代码并写入文件，完成代码的翻译。

### 3.2.4模块间接口

**（1）词法分析与语法分析接口**

语法分析程序通过调用yylex（）函数，从符号串中去一个新的记号，并进行分析。

**（2）语法分析与符号表接口**

/\* 符号表的添加操作 \*/

extern STATUS add\_element(const char\* name, symbol\_type type, symbol\_type sub\_type,

unsigned int dimension, const int is\_const, const int is\_reference);

/\* 函数参数元素在符号表的添加操作 \*/

extern STATUS add\_parameter\_element(const char\* function\_name, const char\* name, symbol\_type type, symbol\_type sub\_type,

unsigned int dimension, const int is\_const, const int is\_reference);

/\* 符号表的定位操作 \*/

extern STATUS locate\_table(void);

/\* 符号表的重定位操作 \*/

extern STATUS relocate\_table(void);

/\* 返回符号的类型信息 \*/

extern symbol\_type get\_type(const char\* name);

/\* 返回符号的返回值类型信息 \*/

extern symbol\_type get\_return\_type(const char\* name);

/\* 返回函数声明的形参个数 \*/

extern unsigned int get\_parameter\_number(const char\* name);

/\* 判断一个符号是不是常量 \*/

extern int is\_const(const char\* name);

/\* 判断一个符号是不是引用 \*/

extern int is\_reference(const char\* name);

/\* 用于判断一个函数参数是不是引用 \*/

extern int is\_reference\_parameter(const char\* function\_name, const char\* name);

/\* 比较函数参数类型 \*/

extern STATUS check\_parameter\_types(const char\* function\_name, const int n, const symbol\_type st[]);

/\* 检查符号是否被使用 \*/

extern int check\_element(const char\* name);

接口如上，描述如注释所述。

# 4．详细设计

## 4.1 词法分析详细设计

### 4.1.1 接口描述

通过yylex（）函数给语法分析程序输出记号。

### 4.1.2 功能描述

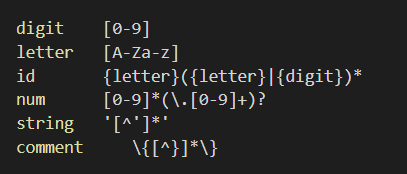
词法分析主要任务是从左到右逐个字符地对源程序进行扫描，按照Pascal语言的词法规则识别出一个个单词符号，产生用于语法分析的记号序列，并把来自编译程序的错误信息和源程序联系起来，记住单词在源程序中的行/列位置，从而行号可以作为错误信息的一部分提示给用户。

源程序中的关键字（除开头的program和末尾的end之外）前、后必须有空格符或换行符，其它词汇间的空格符是可选的。源程序中的注释：用一对花括号括起来，可以出现在任何单词之后。编译程序应该可以处理注释。

如果词法分析程序从源程序中读入了一个不合法的字符即Pascal语言不包括此字符开头的单词符号。词法分析显示打印错误信息，并跳过这个字符，然后继续识别和分析下一个单词符号。

### 4.1.3 所用数据结构说明

设置了如下正则式进行匹配，其中id匹配标识符，num匹配数字，comment匹配注释



### 4.1.4 算法描述

通过正则式匹配要匹配的标识符，完成一定动作（行号、列号++、yylval=目标值），并返回标记值

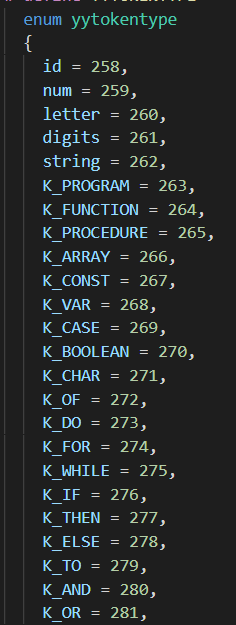
## 4.2 语法分析详细设计

### 4.2.1 接口描述

### 4.2.2 功能描述

利用Pascal语言的语法规则，对词法分析程序产生的记号流进行规约，检查程序的结构是否正确。并且因为采用语法制导翻译技术，在规约的过程中实现语义分析和代码生成。并对语法分析中出现的错误进行处理。

### 4.2.3 所用数据结构说明



Yacc自动为每个标记生成标号

### 4.2.4 算法描述

**LR分析法算法描述：**

do {

令S是栈顶状态，a是ip所指向的符号

if (action[S，a]==shift S′) {

把a和S′分别压⼊入符号栈和状态栈的栈顶;

推进ip，使它指向下⼀一个输⼊入符号;

};

else if (action[S，a]==reduce by A→β) {

从栈顶弹出|β|个符号;

令S′是现在的栈顶状态，把A和goto[S′，A]分别

压⼊入符号栈和状态栈的栈顶;

输出产⽣生式A→β;

};

else if (action[S，a]==accept) return;

else error();

} while(1)；

## 4.3 符号表详细设计

### 4.3.1 接口描述

/\* 符号表的添加操作 \*/

extern STATUS add\_element(const char\* name, symbol\_type type, symbol\_type sub\_type,

unsigned int dimension, const int is\_const, const int is\_reference);

/\* 函数参数元素在符号表的添加操作 \*/

extern STATUS add\_parameter\_element(const char\* function\_name, const char\* name, symbol\_type type, symbol\_type sub\_type,

unsigned int dimension, const int is\_const, const int is\_reference);

/\* 符号表的定位操作 \*/

extern STATUS locate\_table(void);

/\* 符号表的重定位操作 \*/

extern STATUS relocate\_table(void);

/\* 返回符号的类型信息 \*/

extern symbol\_type get\_type(const char\* name);

/\* 返回符号的返回值类型信息 \*/

extern symbol\_type get\_return\_type(const char\* name);

/\* 返回函数声明的形参个数 \*/

extern unsigned int get\_parameter\_number(const char\* name);

/\* 判断一个符号是不是常量 \*/

extern int is\_const(const char\* name);

/\* 判断一个符号是不是引用 \*/

extern int is\_reference(const char\* name);

/\* 用于判断一个函数参数是不是引用 \*/

extern int is\_reference\_parameter(const char\* function\_name, const char\* name);

/\* 比较函数参数类型 \*/

extern STATUS check\_parameter\_types(const char\* function\_name, const int n, const symbol\_type st[]);

/\* 检查符号是否被使用 \*/

extern int check\_element(const char\* name);

### 4.3.2 功能描述

检查语义正确性并辅助生成代码

### 4.3.3 所用数据结构说明

/\* 符号元素结构体 \*/

typedef struct \_symbol

{

    const char\* name; // 符号名

    symbol\_type type; // 符号类型

    symbol\_type sub\_type; // 子类型

    unsigned int dimension; // 维数

    int is\_const; // 是否常量

    int is\_reference; // 是否引用

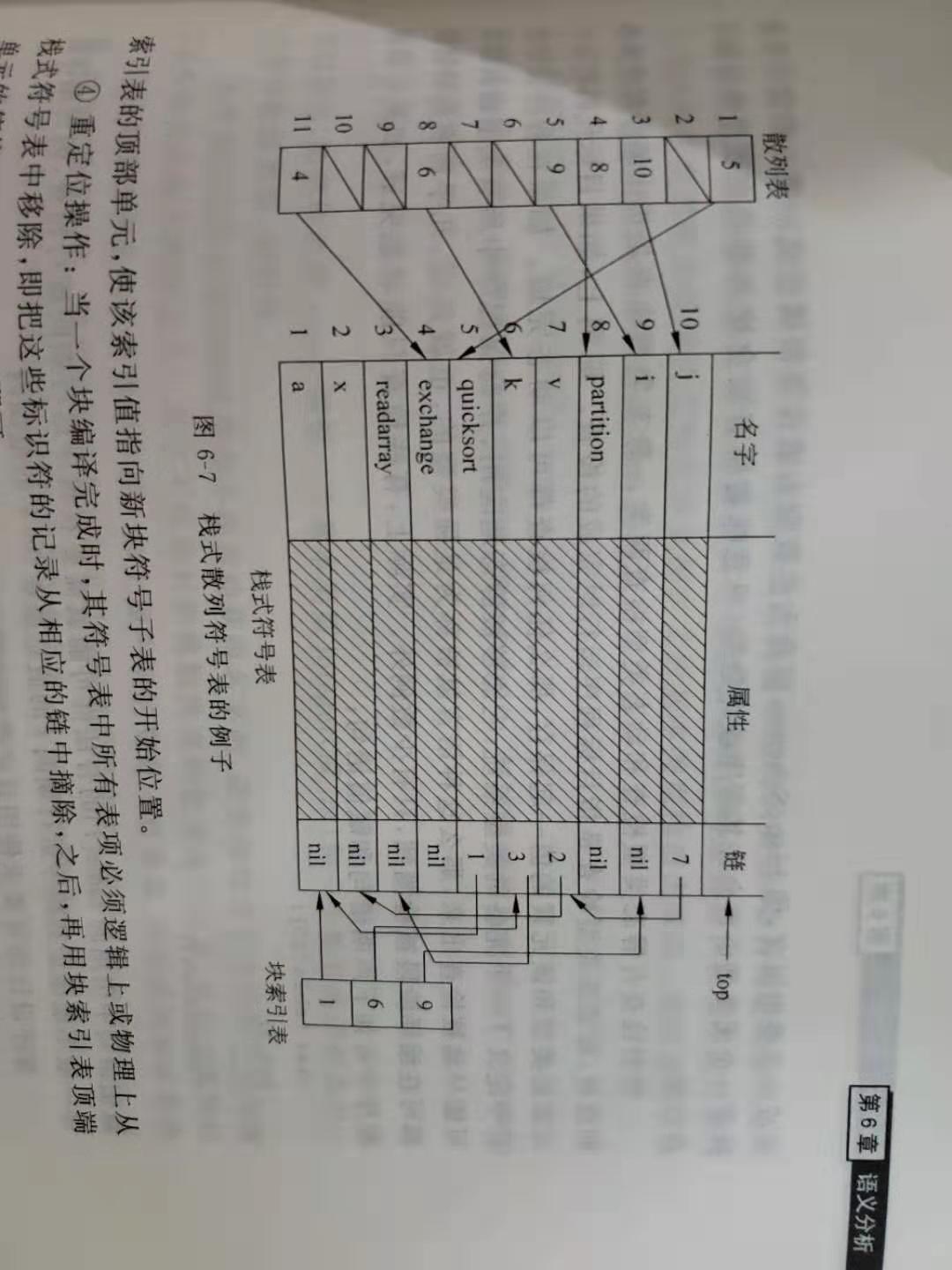
    parameter\_info parameter\_list; // 函数参数

    unsigned int next; // 链域

} symbol;

### 4.3.4 算法描述

在本代码中，根据课本上的讲授，符号表由三个部分组成：散列表，符号栈，程序块表。



**散列表**

散列表使用了一个一维数组作为散列表存储hash值，以符号的首字母作为键，因此散列表中最多存储52个键值，对于大小写的26个字母。同时在符号表中设置链域，构建检错链来解决散列表的冲突问题

**符号栈**

符号栈使用了一个结构体数组进行存储，在结构体中存储了符号名，符号类型，符号维数，符号子类型（即数组存储的值的类型，函数的返回值类型）。

**程序块表**

程序块表使用一个一维数组实现了一个栈，栈顶存储的是当前程序块中的符号在符号表中的开始位置。

**算法核心**

当遇见一个新的程序块时，进行符号栈的定位操作，将当前符号表栈顶的下标压入程序块中。遇到符号后，把符号压入符号栈中，用于进行语义和语法分析。

当一个程序块结束，返回原来程序块时，进行重定位操作，将程序块表进行弹栈，然后对符号表进行弹栈操作至程序块表栈顶指向的位置。

这样，在进行代码编译的时候就可以动态地维护一张符号表，当进行语义分析时可以非常轻松地进行获取目标符号的符号名，符号类型，符号的维数（数组大小，可以进行数组越界检查）。

## 4.4 语义分析详细设计

### 4.4.1 接口描述

调用符号表的add\_element和接口，向符号表中插入变量名、常量名、函数名，插入时会判断是否重定义；

调用符号表的add\_parameter\_element接口，向符号表中插入函数参数；

调用符号表的check\_element接口，判断变量、常量、函数是否未定义；

调用符号表的get\_type接口，查询变量、常量的类型；

调用你符号表的get\_return\_type接口，查询你数组元素、函数返回值的类型；

### 4.4.2 功能描述

实现类型检查，包括变量、数组元素、函数返回值以及它们组合成的表达式之间的类型比较检查；实现变量、常量、函数的重定义和未定义检查；实现数组下标合法性检查。

### 4.3.3 所用数据结构说明

symbol\_type left\_type; *//赋值符左类型*

symbol\_type right\_type; *//赋值符右类型*

symbol\_type exp\_type; *//expression类型*

symbol\_type exp\_left\_type; *//expression左类型*

symbol\_type exp\_right\_type; *//expression右类型*

symbol\_type simexp\_type; *//simple\_expression类型*

symbol\_type simexp\_left\_type; *//simple\_expression左类型呢*

symbol\_type simexp\_right\_type; *//simple\_expression右类型*

symbol\_type term\_type; *//term类型*

symbol\_type term\_left\_type; *//term左类型*

symbol\_type term\_right\_type; *//term右类型*

symbol\_type factor\_type; *//factor类型*

symbol\_type variable\_type; *//variable类型*

### 4.3.4 算法描述

对每种需要类型的语法单位设置全局变量type, left\_type, right\_type，记录当前规约的类型信息。

statement:

variable

{

left\_type = variable\_type;

}

assignop

expression

{

right\_type = exp\_type;

check(left\_type, right\_type);

}

expression:

simple\_expression

relop

{

exp\_left\_type = simexp\_type;

}

simple\_expression

{

exp\_right\_type = simexp\_type;

exp\_type = check(left\_type, right\_type);

}

|

simple\_expression

{

exp\_type = simexp\_type;

}

simple\_expression:

{

simexp\_left\_type = simexp\_type;

}

addop term

{

simexp\_right\_type = term\_type;

simexp\_type = check(simexp\_left\_type, simexp\_right\_type);

}

|

{

simexp\_type = term\_type;

}

term:

term

{

term\_left\_type = term\_type;

}

mulop factor

{

term\_right\_type = factor\_type;

term\_type = check(term\_left\_type, term\_right\_type)

}

|

factor

{

term\_type = factor\_type;

}

factor:

num

{

factor\_type = float\_or\_int($1);

}

|

variable

{

factor\_type = variable\_type;

}

id

{

factor\_type = get\_return\_type($1)

}

P\_LPAREN expression\_list P\_RPAREN

{

factor\_type = get\_return\_type($1); //函数返回类型

}

|

P\_LPAREN expression P\_RPAREN

{

factor\_type = exp\_type;

}

variable:

id

{

variable\_type = get\_type($1);

}

|

id

{

variable\_type = get\_return\_type($1); //数组

} id\_varpart

## 4.5 代码生成

### 4.5.1 接口描述

content=char\_strcpy(content, temp);

将当前翻译得到的temp拼接到已经翻译的content里，扫描完程序后，content里保存了完整的生成的c语言代码。

### 4.5.2 功能描述

将pascal代码翻译成c代码，保存在文件里

### 4.5.3 所用数据结构

char \*content;

### 4.5.4 算法描述

在产生式嵌入的语义动作中，根据Pascal和C的语法单位的对应规则，对于Pascal和C中顺序不一致的部分，先用全局变量保存识别的信息，当规约部分可以与C语言的语法单位对应时，输出它在C语言中对应的句子（保存到字符串content中）。

以const声明为例，pascal中const声明为const a, b = 2; 在C语言中对应为const int a=2; const int b=2;我们需要在const\_vale规约时将类型const\_value的类型用你全局变量const\_type记录下，然后规约const a, b=2;时将刚刚记录的const\_type提前，即const int a; const int b;

const\_declaration:

const\_declaration P\_SEMICOLON

{

content=char\_strcpy(content, ";");

}

id O\_EQL const\_value

{

if (add\_element($4, str2symtype(const\_type), 0, 0, 1, 0) == FAILED) {

yyerror(char\_strcpy($4, " redeclaration！"));

}

content = char\_strcpy(content, "const ");

content = char\_strcpy(content, const\_type);

content = char\_strcpy(content, " ");

content = char\_strcpy(content, $4);

content = char\_strcpy(content, "=");

content = char\_strcpy(content, const\_result);

const\_result = NULL;

*/\* const\_declaration → id = const\_value \*/*

}

|

id O\_EQL const\_value

{

if (add\_element($1, str2symtype(const\_type), 0, 0, 1, 0) == FAILED) {

yyerror(char\_strcpy($1," redeclaration！"));

}

content = char\_strcpy(content, "const ");

content = char\_strcpy(content, const\_type);

content = char\_strcpy(content, " ");

content = char\_strcpy(content, $1);

content = char\_strcpy(content, "=");

content = char\_strcpy(content, const\_result);

const\_result = NULL;

*/\* const\_declaration → id = const\_value \*/*

}

const\_value:

O\_PLUS num

{

if (strchr($2, '.')) {

const\_type = "float";

} else {

const\_type = "int";

}

const\_result = char\_strcpy("+", $2);

*/\* const\_value → + num \*/*

}

其他语法单位的识别基本都使用了这种思想。

# 5．源程序清单

词法分析程序：lex.l

语义和语法分析程序：codeanalys.y

符号表头文件：SymbolTable.h

符号表程序：SymbolTable.c

辅助API：util.c

# 6．程序测试

## 6.1 测试环境：

Linux version 4.4.0-18362-Microsoft (Microsoft@Microsoft.com) (gcc version 5.4.0 (GCC) )（Windows 10的Linux子系统: Windows Subsystem Linux）

bison (GNU Bison) 3.0.4

flex 2.6.4

## 6.2 测试的功能

### 6.2.1通用代码测试

program example(input,output);

const d = +2; e = -2;

var x, y:integer;

z:real;

temp:integer;

a, b, c:char;

h: array[0..10] of integer;

i: array[0..10] of integer;

j: array[0..10] of integer;

k: array[0..20, 0..30] of integer;

m: array[0..10] of char;

procedure init(a:integer);

begin

if a < 5 then

a := a + 1;

for temp := 0 to 9 do

begin

h[temp] := 2\*temp + 2;

i[temp] := (temp + a) \* 2;

end;

end;

procedure print(a:integer);

begin

for temp := 0 to 9 do

writeln(j[temp]);

end;

function gcd(a,b:integer):integer;

begin

if b=0 then exit(a)

else exit(gcd(b, a mod b))

end;

begin

init(3);

temp := 0;

while temp <= 9 do

begin

j[temp] := gcd(h[temp], i[temp]);

temp := temp + 1;

end;

print(1);

end.

这是由我们自己编写的GCD例程升级版程序，其中测试了包括函数调用、变量的定义、二维数组的定义、类型检查等等。通过运行我们的代码，实现了从Pascal到C的转换。

#include<stdio.h>

#include<stdlib.h>

const int d=+2;const int e=-2;

int x,y;

float z;

int temp;

char a,b,c;

int h[10];

int i[10];

int j[10];

int k[20][30];

char m[10];

void init (int a )

{

{

if(a<5)

a=a+1;

for(temp=0;temp<=9;temp++)

{

h[temp]=2\*temp+2;

i[temp]=(temp+a)\*2;

}

}

}

void print (int a )

{

{

for(temp=0;temp<=9;temp++)

printf(j[temp]);

}

}

int gcd (int a,int b )

{

{

if(b==0)

return(a);

else

return(gcd(b,a%b));

}

}

int main()

{

init(3);

temp=0;

while(temp<=9){

j[temp]=gcd(h[temp],i[temp]);

temp=temp+1;

}

print(1);

return 0;

}

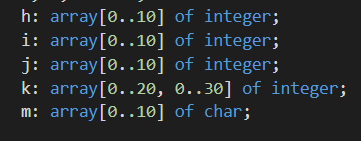
转换后的C语言

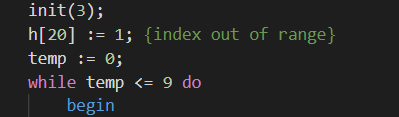
将printf(j[temp])手动改成printf(“%d, ”, j[temp])后，可以通过gcc编译，执行后得到的结果也正确。

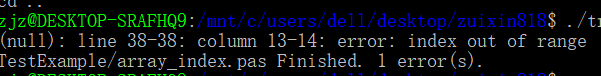


### 6.2.2数组越界测试

在这个测试中，我们定义的数组是[0…10],但我们给h[20]赋值，进行测试。



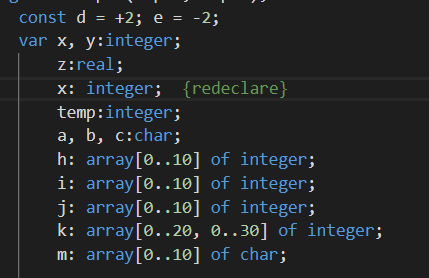


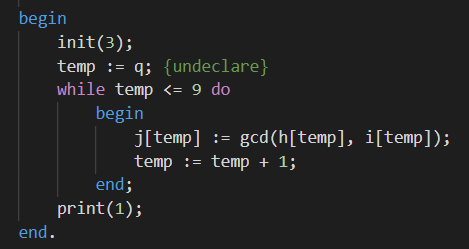


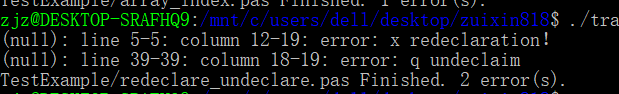
我们会发现编译程序会报错index out of range，并且给出了出错行号列号。

### 6.2.3重复定义未定义测试

在这个测试中我们进行了重复定义变量和未定义变量测试。



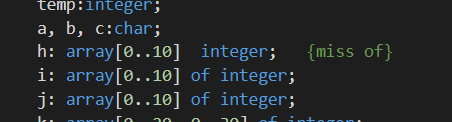




我们发现我们的程序对于redeclaration 和 undeclare都给出了报错。

### 6.2.4语法错误测试

在这一部分，我们通过删除一些关键词，使不能用Yacc中的产生式匹配到，进行测试。





我们发现我们的编译器会告诉你，这个地方少了一个Kword of，需要填补上。这个其实是由Yacc底层的一些东西帮助我们进行实现的。

# 7．课程设计总结

## 7.1 体会与收获：

**张帅帅**：作为此次编译原理课程设计的组长，因为组队时间较晚，我们的队伍只有3个人，所以感到自己身上的责任重大和编写任务的紧迫。在课程设计中积极和组员协调编码，并扛起了组内最重最难的编写任务。作为组长，我责无旁贷，常常从下午开始编写代码直至中午11 12点，用“沉溺代码之中，不知东方之既白”来描述确切不过了。往往睡2 3 个小时后，又与组员开会，协调晚上代码进度，推动整个项目的进展。

在编写中主要负责了语法分析、语义分析、代码生成部分的主要工作，是全队的核心。

**张津彬**:作为此次编译原理课程设计3人小组的成员，我也在这次本次实验中做了许多工作。因为组长是我们公认的代码大佬，能与组长一起完成项目十分充实、收获很多。我主要负责了包括符号表的插入、查询、重定位等操作，写了与语法分析程序接口的编写等任务。在一开始觉得任务十分繁重，但经过大家的分工与协商，最重一点点分解任务，完成了整个编译器的实现。

**张君智:** 我作为此次3人小组中没两位大佬那么强的同学，我也承担起来了不少编码工作。从开始的理论知识指导实践工作方面，包括选定我们的技术路线，实现方法等，都做出了贡献。我们通过利用语法制导翻译方案，把类型检查、代码生成都放在由语义分析主导的程序中，简化编码，高效完成。在项目工作方面，主要负责了词法分析部分的代码编写、语法分析部分一部分产生式的编写以及整体理论指导。在开始负责了环境的配置，工具的探索等任务，以及最后的调试、测试、ppt制作、文档撰写等工作。这次的实践让我对于编译器的实现等底层核心技术有了更深一步的了解，让我对于理论知识的掌握更加清楚明了。并且在这次的实际编写编译器的过程中极大的锻炼了我的动手能力、编码能力、实践能力。

## 7.2设计过程中遇到或存在的主要问题及解决方案

遇到的问题：时间紧、任务重，团队内部产生矛盾。在代码方面开始不太理解lex和yacc的使用方法，后来不断摸索，基本有了很好的理解。

## 7.3 改进建议

可以尝试更多、更新鲜、更有趣、更接近实际的任务让同学们去做、最近正值中美贸易战，深刻感受到了很多核心、底层技术方面，我们和别人的差距。老师们可以去探索更多目前实际用到的一些编译器实现的工作等，让我们进行更多更有趣的尝试。