《编译技术》

课程设计文档

学号：\_\_\_\_\_13071079\_\_\_\_\_

姓名：\_\_\_\_\_\_\_卢嵩\_\_\_\_\_\_\_

2016年 11 月 14 日

# 需求说明

## 文法说明

### 获取的完整文法

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜＜非零数字＞

＜非零数字＞ ::= １｜．．．｜９

＜字符＞ ::= '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'

＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"

＜程序＞ ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}

| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’ )}

＜常量＞ ::= ＜整数＞|＜字符＞

＜类型标识符＞ ::= int | char

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’

＜无返回值函数定义＞ ::= void＜标识符＞‘(’＜参数＞‘)’‘{’＜复合语句＞‘}’

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数＞ ::= ＜参数表＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜因子＞ ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’|‘(’＜表达式＞‘)’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;

|＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;|＜情况语句＞｜＜返回语句＞;

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞＜缺省＞ ‘}’

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

＜情况子语句＞ ::= case＜常量＞：＜语句＞

＜缺省＞ ::= default : ＜语句＞|＜空＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

＜语句列＞ ::= ｛＜语句＞｝

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

＜写语句＞ ::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’| printf ‘(’＜字符串＞ ‘)’| printf ‘(’＜表达式＞‘)’

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

附加说明：

（1）char类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符

（2）标识符区分大小写字母

（3）写语句中的字符串原样输出

（4）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句

（5）数组的下标从0开始

### 文法改写

为简化设计编译器，本人对原始文法进行了部分改写，主要改写部分采用红色加粗表示。改写后的文法如下：

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜＜非零数字＞

＜非零数字＞ ::= １｜．．．｜９

＜字符＞ ::= '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'

＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"

＜程序＞ ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}

| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜声明头部＞ ::= **＜类型标识符＞** ＜标识符＞

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’ )}

＜常量＞ ::= ＜整数＞|＜字符＞

＜类型标识符＞ ::= int | char

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜**参数表**＞‘)’ ‘{’＜复合语句＞‘}’

＜无返回值函数定义＞ ::= void＜标识符＞‘(’＜**参数表**＞‘)’‘{’＜复合语句＞‘}’

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜因子＞ ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’|‘(’＜表达式＞‘)’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;

|＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;|＜情况语句＞｜＜返回语句＞;

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞＜缺省＞ ‘}’

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

＜情况子语句＞ ::= case＜常量＞：＜语句＞

＜缺省＞ ::= default : ＜语句＞|＜空＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

＜语句列＞ ::= ｛＜语句＞｝

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

＜写语句＞ ::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’| printf ‘(’＜字符串＞ ‘)’| printf ‘(’＜表达式＞‘)’

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

改动总结：删除<参数>项，使用＜参数表＞代替＜参数＞，头部声明修改

### 文法扩充说明：

为了输出美观，printf结束后默认换行。

## 目标代码说明

要生成的目标代码为32位MIPS汇编代码，具体包含的指令如下，注意这里的指令包括部分MARS支持的伪指令：

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 助记符 | 示例 | 含义 |
| 加 | add | add $1,$2,$3 | $1 = $2 + $3 |
| 减 | sub | sub $1,$2,$3 | $1 = $2 - $3 |
| 乘 | mult | mult $1, $2 | Hi,Lo = $1 \* $2 |
| 除 | div | div $s, $t | $LO = $s / $t;  $HI = $s % $t; |
| 立即数加 | addi | addi $t, $s, 1406 | $t = $s + 1406 |
| 立即数减 | subi | subi $t, $s, 1406 | $t = $s - 1406 |
| 取字 | lw | lw $t, 4($s) | $t = MEM[$s + 4] |
| 存字 | sw | sw $t, offset($s) | MEM[$s + offset] = $t; |
| 条件分支 | beq | beq $s, $t, offset | if $s == $t advance\_pc (offset << 2)) ; else advance\_pc (4); |
| 条件分支 | bne | bne $s, $t, offset | if $s != $t advance\_pc (offset << 2)) ; else advance\_pc (4); |
| 无条件跳转 | j | j target | PC = nPC; nPC = (PC & 0xf0000000) | (target << 2); |
| 跳转并返回 | jr | Ja $ra | Goto $ra |
| 系统调用 | syscall | syscall | 执行相关系统函数 |

## 优化方案\*

常数合并，DAG消除基本块内部的公共子表达式，寄存器分配优化，数据流分析，循环优化等

# 详细设计

## 程序结构

程序大体上分为8个文件，其中有7个cpp文件，一个头文件。具体结构如下图，执行的顺序由箭头所示。其中global.h中包含了所有宏定义以及所有执行所需要用到的函数定义。程序接受到输入后首先执行词法分析程序，词法分析完成以后执行语法分析，确定输入的语言成分，完成建立符号表等相关操作。在进行词法和语法分析的过程中可能会调用error.h来进行错误信息的处理。在完成语法分析之后，程序将会生成中间代码，中间代码以四元式的形式表示，具体的四元式形式见后文。在完成四元式生成之后，我们的程序将会把四元式转化为mips汇编程序并做输出，完成编译工作。在进行中间代码生成和汇编代码生成的时候需要同时兼顾代码的优化，尽量提高生成代码的质量。



Figure 编译程序结构图

## 类/方法/函数功能

各类的功能如上图所示，再次不做赘述。以下列举全部函数及其功能：

### 词法分析：

|  |  |
| --- | --- |
| 函数 | 功能 |
| void getpath() | 用来获取输入文件的路径 |
| void clearToken() | 用来清空token |
| void getnextline() | 获取下一行源代码的输入 |
| void getch() | 获得下一个输入字符，当读到文件末尾的时候返回空字符 |
| void catToken() | 将当前字符与token拼接 |
| void skip() | 跳过无意义的字符，本文法中针对空格，换行符和制表符 |
| bool isLetter() | 判断当前ch是否是字母类型，注意根据文法定义下划线'\_'也算作字母 |
| bool isDigit() | 判断ch是否是数字 |
| void retract() | 读字符的指针退格 |
| int isreserve() | 分辨当前标识符是否是保留字，是的话返回对应的sym，不是返回0说明是标识符 |
| bool isStringCon() | 检查当前ch是不是组成字符串的合法元素，文法定义为“十进制编码为32,33,35-126的ASCII字符” |
| int getsym() | 词法分析主要部分，判别当前单词的类型 |
| void output() | 将分析结果输出，为下一步语法分析做准备 |

### 语法分析

|  |  |
| --- | --- |
| 函数 | 功能 |
| void enter(); | 将符号存入符号表 |
| void program(); | 处理程序的递归子程序 |
| void condecl(); | 处理常量说明的递归子程序 |
| void constdef(); | 处理常量定义的递归子程序 |
| void vardecl(); | 处理变量声明的递归子程序 |
| void vardef(); | 处理变量定义的递归子程序 |
| void headdef(); | 处理头声明的递归子程序 |
| void paralist(); | 处理函数声明时参数表的递归子程序 |
| void funcdecl(); | 处理函数声明的递归子程序 |
| void statement(); | 处理语句的递归子程序 |
| void expression(); | 处理表达式的递归子程序 |
| void term(); | 处理项的递归子程序 |
| void factor(); | 处理因子的递归子程序 |
| void assignstate(); | 处理赋值语句的递归子程序 |
| void ifstate(); | 语句的递归子程序 |
| void whilestate(); | 处理while循环语句的递归子程序 |
| void switchstate(); | 处理switch的递归子程序 |
| void printfstate(); | 处理printf的底稿子程序 |
| void scanfstate(); | 处理scanf的递归子程序 |
| void returnstate(); | 处理return的递归子程序 |

### 错误处理

|  |  |
| --- | --- |
| 函数 | 功能 |
| void error(int errid) | 根据不同的错误代码来输出不同的错误信息 |
| void test(*vector* <char> &symset) | 词法出错后跳过错误字符 |

### 中间代码生成

|  |  |
| --- | --- |
| 函数 | 功能 |
| void emit(int fctno, int a, int b, int c) | 根据不同的fctno和a,b,c输入来生成四元式 |

### 汇编代码生成

|  |  |
| --- | --- |
| 函数 | 功能 |
| void quad2asm(); | 将生成的四元式转化为汇编代码 |
| void insertasmsym(); | 生成汇编符号表 |
| void findsym(const *string* &); | 找到符号在符号表中的位置 |
| void pushstack(int len); | 进行压栈操作 |
| void setlab(); | 生成新的label |
| void jumpasm(); | 生成跳转指令 |
| void assignasm(); | 生成赋值语句 |
| void ifasm(); | 生成条件语句 |
| void switchasm(); | 生成分支语句 |
| void constdeclasm(); | 常量定义语句 |
| void vardeclasm(); | 变量定义语句 |
| void callasm(); | 函数调用语句 |
| void paraasm(); | 参数处理语句 |
| void scanfasm() | 处理scanf |
| void printfasm() | 处理printf |
| void returnasm() | 生成return语句 |

### 代码优化

|  |  |
| --- | --- |
| 函数 | 功能 |
| void killduplicate(); | 消除子表达式 |
| void constfolding(); | 常熟合并 |
| void registeropt(); | 寄存器优化 |
| void loopopt(); | 循环优化 |

## 调用依赖关系



## 符号表管理方案

符号表拟采用栈式存储结构，采取先入后出的管理算法，具体的数据结构如下所示

#define TABLESIZE 1000 //符号表大小

#define MAXLAYER 100 //符号表最大层次

typedef struct //定义表项

{

*string* name; //符号的名字

int addr; //符号的地址

int kind; //符号的大类别

int type; //符号的数据类型

int value; //符号的值

int number; //参数或者数组中元素的个数

} tableitem;

typedef struct //定义符号表

{

tableitem item[TABLESIZE]; //建立数组存储单个表项

int top; //符号表标顶位置

}symtable;

## 存储分配方案

【说明运行时的存储组织及管理方案，运行栈结构】

### 存储组织

运算时需要的变量等数值均存储于计算器中，采用FIFO寄存器分配策略，需要保存的变量存储在.data区中。

### 运行栈结构

假设运行到main函数中调用了某个函数，那么运行栈的结构大致如下表所示：

|  |
| --- |
| 栈顶 |
| f.局部数据区 |
| f.参数区 |
| f.prevabp |
| f.ret |
| f.abp |
| Main.abp |
| O.S. |
| Main.局部数据区 |
| Main.prevabp |
| Main.ret |
| main.abp |
| O.S. |
| 函数声明区 |
| 全局变量区 |
| 全局常量区 |
| O.S. |

## 解释执行程序\*

本编译器的目的为生成目标代码，无解释执行程序。

## 四元式设计\*

四元式拟采用类似于符号表的存储结构，先通过语法分析得到sym，之后在根据语法分析得到的语义生成四元式并储存起来，为后面的目标代码生成做准备。

|  |  |
| --- | --- |
| 四元式 | 含义 |
| Con, a, b, c | Const a b = c, 其中a为类型，b为常量名，c为常量值 |
| Int, a, , | Int a;其中a为变量名 |
| Ch, a, , | Char a; 其中a为变量名 |
| Para, a, b, | Para a是参数类型，数据类型为b |
| =, a, b, | a = b |
| []，a, b , c | c = a[b] |
| Arr=, a, b, c | a[b] = c |
| ==, a,b,c | C = (a == b) |
| <=,a,b,c | C= (a<=b) |
| <,a,b,c | C = (a <b) |
| Func, a, , | F() |
| Valuef, , ,a | A= f() |
| Jump, , ,label | Jmp label |
| Jal, , ,label | Jal label |
| Lab, , ,a | Set label a |
| Startf, a, f, | 开始函数f，返回值为a |
| Endf, f, , | 结束函数f |
| Scan,a, , | Scanf(a) |
| Print,a,b, | Printf(a,b) |
| Ret,a | Return(a) |

## 目标代码生成方案\*

目标代码生成需要同时运用到符号表symtable以及四元式表quadtable。当前的考虑是现根据符号表中符号所在栈层次和quadtable中符号的位置来确定生成汇编符号的名字。例如当main中和main调用的函数f（）中都存在一个叫做a的变量的话，f()中a 的名字就需要被其他变量名所替代，否则生成的汇编代码就会出现混乱。在生成目标代码的同时需要不断地兼顾优化。由于中间代码属于上下文无关代码，因此生成目标代码的时候只需要进行目标代码与四元式的一一对应即可。

|  |  |
| --- | --- |
| Con, a, b, c | If a == int, .data b: .word c  Else .data prime: b: .asciiz c |
| Int, a, , | 给a分配一个空闲的临时寄存器 |
| Ch, a, , | 给a分配一个空闲的临时寄存器 |
| Para, a, b, | 给b分配一个参数寄存器 |
| =, a, b, | a : $t1  li $t1, b |
| []，a, b , c | a : $t1 c:$t3  lw $t2,b($t1)  addi $t3,$t2,0 |
| Arr=, a, b, c | a: $t1 c:$t3  la $t2,b($t1)  sw $t3,0($t2) |
| ==, a,b,c | a: $t1 b:$t2 c:$t3  sub $t4, $t1,$t2  beqz $t4,EUQ  j END  EQU:  addi $t3,$0,1  END: |
| <=,a,b,c | a: $t1 b:$t2 c:$t3  sub $t4, $t1,$t2  blez $t4,EUQ  j END  EQU:  addi $t3,$0,1  END: |
| <,a,b,c | a: $t1 b:$t2 c:$t3  sub $t4, $t1,$t2  bltz $t4,EUQ  j END  EQU:  addi $t3,$0,1  END: |
| Func, a, , | j a |
| Valuef, , ,a | jal a |
| Jump, , ,label | j label |
| Jal, , ,label | jal label |
| Lab, , ,a | a : |
| Startf, a, f, | BEGINF: |
| Endf, f, , | ENDF: |
| Scan,a, b, | a : $t1 b : $t2.  bgtz $2, CHAR  li $v0, 5  syscall  j END  CHAR:  li $v0, 12  syscall  add $t1,$0,$v0 |
| Print,a,b, | li $v0, 10  syscall |
| Ret,a | addi $v0,$0,a  jr |

## 优化方案\*

### 常数合并

通过窥孔优化进行常数合并操作，减少查表时间。

### 消除公共子表达式

采用书上的启发性算法实现公共子表达式的删除。

### 数据流分析

通过划分基本块，并使用数据流分析的方法，合理分配临时寄存器。在编译MIPS程序时，我们将$t0~$t9视作临时寄存器，如果函数中的变量多于10个，那么多于的变量将会存储到data数据区中。

## 出错处理

错误处理选择在词法分析和语法分析的时候进行，确保生成四元式的过程都是没有问题的代码。错误处理嵌入到词法和语法分析的代码中，error.cpp中的error()函数负责输出对应的错误信息，test函数用来跳过不合法的符号集，直到找到合法的符号集再重新开始执行词法分析程序。错误编号如下：

|  |  |  |
| --- | --- | --- |
| 错误类型 | 错误编号 | 错误说明 |
| UNDEF\_ID | 1 | 未声明标识符 |
| MULTI\_DEF | 2 | 重定义标识符 |
| IDENT\_TOO\_LONG | 3 | 标识符过长 |
| PROGRAM\_ERROR | 4 | 程序结构错误 |
| STR\_CONTENT\_ERROR | 5 | 字符串内容错误 |
| STR\_TOO\_LONG | 6 | 字符串超长 |
| ZERO\_HEAD\_NUM | 7 | 非零数字以0开头 |
| NUM\_HEAD\_IDENT | 8 | 数字开头的标识符 |
| NUM\_TOO\_LARGE | 9 | 数字太大 |
| UNDEF\_INPUT | 10 | 不合法输入 |
| CHAR\_MISS\_QUOTE | 11 | 字符类型丢失后一个单引号 |
| CHAR\_CONTENT\_ERROR | 12 | 字符内容不合法 |
| CHAR\_OVERFLOW | 13 | 字符溢出 |
| NEQUAL\_MISS | 14 | 错误使用不等号，只有！没有等号 |
| MISSING\_SEMICOLON | 15 | 丢失分号 |
| MISSING\_RPARENT | 16 | 丢失右括号 |
| MISSING\_RBRACK | 17 | 丢失右中括号 |
| MISSING\_RBRACE | 18 | 丢失右大括号 |
| MISSING\_IDENT | 19 | 丢失标识符 |
| ERROR\_PARA\_NUM | 20 | 参数数目错误 |
| MISSING\_LPARENT | 21 | 丢失左括号 |
| MISSING\_LBRACK | 22 | 丢失左中括号 |
| MISSING\_LBRACE | 23 | 丢失左大括号 |
| MISSING\_PLUS | 24 | 丢失加号或减号 |
| MISSING\_MULTI | 25 | 丢失乘号或除号 |
| ASSIGN\_ERROR | 26 | 赋值时发生错误 |
| RETURN\_ERROR | 27 | 返回值错误 |
| MISSING\_MAIN | 28 | 丢失main函数 |
| MISSING\_RETURN | 29 | 丢失return语句 |
| EXPRESSION\_ERROR | 30 | 表达式错误 |
| CONST\_NOT\_INIT | 31 | 常量使用前未初始化 |
| VAR\_NOT\_INIT | 32 | 变量使用前未初始化 |
| FUNC\_NO\_RET | 33 | 无返回值函数当做表达式使用 |
| OUT\_OF\_TABLE | 34 | 符号表满了 |
| OUT\_OF\_ARRAY | 35 | 数组溢出 |
| WRONG\_TYPE | 36 | 声明类型错误 |

错误处理方案：

1. 不做处理
2. 跳读到下一个双引号位置
3. 跳读到下一个逗号位置
4. 跳读到下一个括号位置
5. 跳读到某一元素集合的位置
6. 跳读到下一个类型标识符或void的位置
7. 跳读到下一语句列
8. 程序直接退出

# 操作说明

## 运行环境

【说明搭建运行环境的步骤】

编译器可以在windows平台下运行，需要有C++语言的支持。

生成的目标代码可以在MARS上运行。

## 操作步骤

【详细说明操作步骤】

# 测试报告

## 测试程序及测试结果

【给出提供的测试程序以及每个程序的测试结果，至少5个正确程序，5个错误程序，无需截屏】

## 测试结果分析

【说明上述测试程序对语法成分的覆盖情况】

# 总结感想

【说明在完成课程设计中的收获、认识和感想】

注：【】内的文字为文档模板说明，完成的作业中需去掉。

标\*的章节需根据题目的难度进行取舍。