《编译技术》

课程设计文档

学号：\_\_\_\_\_16061075\_\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_黄霁昀\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2018年 11月 17日

# 一．需求说明

## 1．文法说明

### （1）获取的文法：

＜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＞｝｜０

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜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＞']') } //＜无符号整数＞表示数组元素的个数，其值需大于0

＜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＞} //[+|-]只作用于第一个<项>

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

＜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＞;｜＜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＞［else＜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＞ ::= do＜语句＞while '('＜条件＞')' |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞

＜步长＞::= ＜无符号整数＞

＜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＞')']

### （2）改写的文法：

＜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＞｝｜０

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜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＞']') } //＜无符号整数＞表示数组元素的个数，其值需大于0

＜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＞} //[+|-]只作用于第一个<项>

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

＜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＞;｜＜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＞［else＜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＞ ::= do＜语句＞while '('＜条件＞')' |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞

＜步长＞::= ＜无符号整数＞

＜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＞')']

改写位置：使用类型标识符代替int char。

## 2．目标代码说明

要生成的目标代码为32位MIPS汇编代码，具体指令及含义如下：

|  |  |
| --- | --- |
| 指令 | 含义 |
| add | 寄存器之间加运算 |
| addi | 寄存器和立即数加运算 |
| sub | 寄存器之间减运算 |
| subi | 寄存器与立即数减运算 |
| mul | 乘法 |
| div  mflo  mfhi | 除法 |
| sw | 寄存器值保存到内存 |
| lw | 内存值加载到寄存器 |
| beq, bne, bgez, bgtz, blez, bltz | 条件跳转（==/!=/>=/>/<=/<） |
| j jal jr | 跳转（无条件、跳转并存入寄存器、返回） |
| li | 装载立即数到寄存器 |
| move | 将寄存器的值存到另一个寄存器 |
| la | 将标签的地址值赋给寄存器 |
| syscall | 系统调用 |

## 优化方案\*

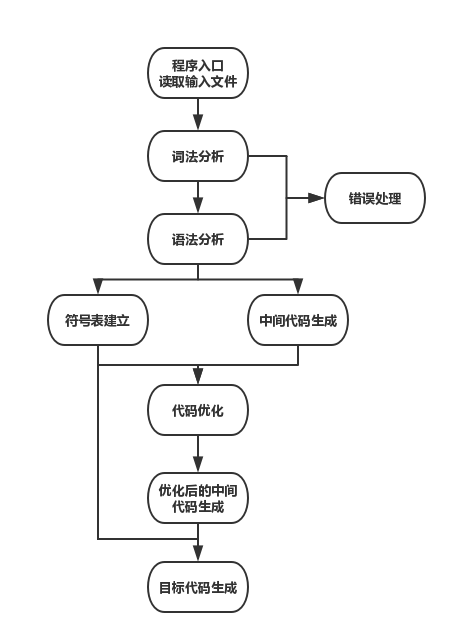
1. 常数合并
2. 基本块内部公共子表达式删除
3. 数据流分析
4. 寄存器分配、调度优化

# 详细设计

注：此设计可能由于后续实际代码编写存在改动，现只为初步框架。可能出现没有考虑周全的部分，同时不够详细的部分会进行后续补充。

## 1．程序结构

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，完成符号表建立和中间代码生成，中间代码以四元式的形式表示，在词法、语法分析的过程中如遇到错误则需要调用错误处理，完成上述过程后，将会进行代码优化并生成优化后的中间代码，最后，程序将会把中间代码转化为mips汇编程序并输出，完成编译工作。

流程如下图：

## 2．类/方法/函数功能

### Lexer（词法分析）

Lexer.cpp Lexer.h

|  |  |
| --- | --- |
| Lexer.cpp  Lexer.h | |
| 方法名 | 功能 |
| getSym():void | 获取sym，并判断类型 |
| output():void | 输出，用于debug |
| init():void | 初始化，获取读入、输出文件等 |
| close():void | 关闭文件流 |
| freshToken():void | 重置token |
| getNextLine():void | 获取下一行 |
| getChar():void | 获取当前字符 |
| skip():void | 跳过空格、换行符、制表符 |
| isLetter():bool | 判断是否为字母 |
| isDigit():bool | 判断是否为数字 |
| constructToken():void | 拼接当前字符到token |
| isReserve():int | 判断是否为保留字并返回保留字类型 |
| isLegalString():bool | 判断是否为合法字符串 |

### Syntax（语法分析）

|  |  |
| --- | --- |
| Syntax.cpp  Syntax.h | |
| 方法名 | 功能 |
| syntax():void | 将源程序转换为中间代码 |
| skip():void | 跳读源程序 |
| createLabel():string | 生成标签名字 |
| constProcess():void | 处理常量声明和定义 |
| varProcess():void | 处理变量声明和定义 |
| factorProcess():string | 处理因子 |
| itemProcess():string | 处理项 |
| expressionProcess():string | 处理表达式 |
| paraProcess():int | 处理参数声明 |
| conditionProcess():int | 处理条件 |
| ifProcess():void | 处理if语句 |
| doWhileProcess():void | 处理do\_while语句 |
| forProcess():void | 处理for语句 |
| assignProcess():void | 处理赋值语句 |
| scanfProcess():void | 处理scanf语句 |
| printfProcess():void | 处理printf语句 |
| returnProcess():void | 处理return语句 |
| statementProcess():void | 处理语句 |
| CompondProcess():int | 处理复合语句 |
| funcDefProcess():void | 处理无返回值函数定义语句 |
| returnFuncDefProcess():void | 处理有返回值函数定义语句 |
| funcProcess():string | 处理无返回值函数调用语句 |
| returnFuncProcess():string | 处理有返回值函数调用语句 |
| mainProcess():void | 处理主函数 |

### （3）Error（错误处理）

|  |  |
| --- | --- |
| Error.cpp  Error.h | |
| 方法名 | 功能 |
| error():void | 错误处理 |

### （4）Enter（登记符号表并输出中间代码）

|  |  |
| --- | --- |
| Enter.cpp  Enter.h | |
| 方法名 | 功能 |
| enter():void | 登记标识符 |
| enterConst():void | 登记常量 |
| enterVar():void | 登记变量 |
| enterArray():void | 登记数组信息 |
| enterFunc():void | 登记函数 |
| enterMidCode():void | 登记中间代码 |
| printTab():void | 打印符号表 |
| printArrayTab():void | 打印数组表 |
| printBlockTab():void | 打印分程序表 |
| printTempMap():void | 打印临时变量表 |
| printMidCode():void | 打印中间代码 |

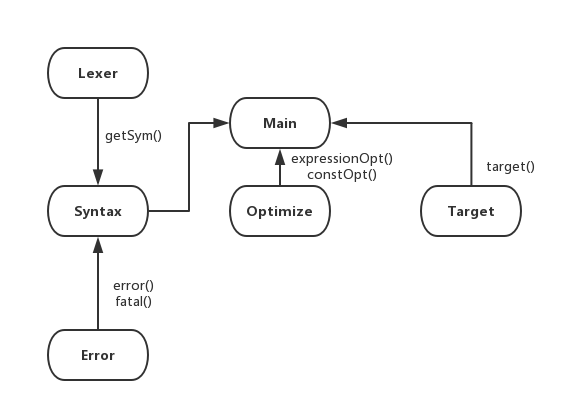
### （6）Optimize（优化）

|  |  |
| --- | --- |
| Optimize.cpp  Optimize.h | |
| 方法名 | 功能 |
| expressionOpt():void | 基本块内公共子表达式删除 |
| isInMap():bool | 是否在节点信息表中 |
| deleteNode():void | 删除节点 |
| getMidNode():Node | 从图中找一个没有父节点的中间节点 |
| findInMap():Node | 在图中寻找节点 |
| ExistInOldNewMap():bool | 检查是否在新旧变量名更新表中 |
| isEqual():Node | 判断是否有等价节点 |
| operandProcess():Node | 处理操作数 |
| resultProcess():Node | 处理结果 |
| insert():void | 插入节点 |
| isOpt():bool | 根据局部变量重定义的结果判断是否进行优化 |
| createOptCode():bool | 生成优化代码 |
| localExpressionOpt():void | 优化局部表达式块 |
| funcOpt():void | 优化函数体 |
| constOpt():void | 优化常量 |

### （7）Target（生成目标代码）

|  |  |
| --- | --- |
| Target.cpp  Target.h | |
| 方法名 | 功能 |
| target():void | 生成目标代码 |
| createLabel():string | 生成标签 |
| pushReg():void | 将当前寄存器值存入运行栈 |
| clearReg()void | 清空寄存器 |
| findInMap():bool | 在map中寻找变量 |
| allocReg():int | 申请寄存器 |
| varCount():void | 记录变量个数 |
| allocRegByCount():void | 根据引用计数分配寄存器 |
| allocRegByFunc():void | 函数入口分配寄存器 |
| OpreandProcess():void | 处理操作数 |
| calProcess():void | 处理加减乘除 |
| assignProcess():void | 处理赋值指令 |
| jumpProcess():void | 处理无条件跳转指令 |
| conditionJumpProcess():void | 处理有条件跳转指令 |
| labelProcess():void | 处理标签 |
| funcProcess():void | 处理函数 |
| inputProcess():void | 处理输入 |
| outputProcess():void | 处理输出 |
| paraProcess():void | 处理参数传递 |
| callProcess():void | 处理函数调用 |
| outputMips():void | 输出目标代码 |

## 3. 调用依赖关系



## 4．符号表管理方案

**struct** Item{  
 string name; //标识符名称   
 **int** kind; //种类(const=1, var=2, func=3, array=4)  
 **int** type; //数据类型(int=1, char=2)  
 **int** addr; //变量的相对地址，常量的具体地址，函数的入口地址  
 **int** ref; //数组在ArrayTable中的位置，函数在BlockTable中的位置  
 **int** link; //上一个标识符在符号表中的位置  
 **int** level; //所在分程序  
};  
  
**struct** Table{//符号表  
 **struct** Item array[MAX]; //符号表数组  
 **int** i; //索引  
};  
  
**struct** ArrayItem{  
 **int** type; //数据类型(int=1, char=2)  
 **int** size; //数组大小  
};  
  
**struct** ArrayTable{//数组表  
 **struct** ArrayItem arrayArray[ARRAYMAX];  
 **int** i; //索引  
};  
  
**struct** BlockItem{  
 **int** last; //最后一个标识符在符号表中的位置  
 **int** lastVar; //最后一个参数在符号表中的位置  
 **int** type; //返回值类型(void=0, int=1, char=2)  
 **int** paraNum; //参数个数  
 **int** varSize; //变量栈空间  
 **int** varNum; //变量个数  
};  
  
**struct** BlockTable{//分程序表  
 **struct** BlockItem blockArray[BLOCKMAX];  
 **int** i; //索引  
};

## 5．存储分配方案

### （1）运行时的存储组织及方案

对于局部变量、常量、形参、全局变量、常量利用符号表储存并利用名字查找到在数据栈中的相对当前数据区的基地址偏移。

对于临时变量建立tempMap表存储，并利用名字查找其所在数据栈地址偏移。

### 运行栈结构

|  |
| --- |
| 临时变量 |
| 局部变量、常量 |
| 参数区 |
| return value |
| prev abp |
| ret addr |
| 临时变量 |
| 局部变量、常量 |
| return value |
| prev abp |
| ret addr(main) |
| 全局变量、常量 |

## 6. 解释执行程序\*

无

## 四元式设计\*

### （1）结构

**struct** quad{  
 **int** op; //四元式操作符  
 string z;  
 string x;  
 string y;  
 **bool** isStart; //标志该四元式中间代码是否为入口语句  
};

### 具体

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| op | z | x | y | 功能 |
| + | z | x | y | add addi |
| - | z | x | y | sub subi |
| \* | z | x | y | mul |
| / | z | x | y | div |
| = | z | (x) | y | move sw lw |
| goto | z |  |  | j |
| beq | z | x | y | beq |
| bne | z | x | y | bne |
| bgez | z | x | y | bgez |
| bgtz | z | x | y | bgtz |
| blez | z | x | y | blez |
| bltz | z | x | y | bltz |
| ret |  |  | (y) | jr |
| jal | z |  |  | jal |
| setlab |  |  | y | la |
| func |  | x | y | 声明函数 |
| para |  | x | y | 函数定义的形参 |
| in |  |  | y | scanf |
| out |  | x | y | printf |
| passPara |  |  | y | 传递参数 |
| call |  |  | y | 函数调用 |

## 8. 目标代码生成方案\*

### （1）代码及代码表

**struct** Code {  
 **int** op; //操作符  
 string z; //标签名或寄存器  
 string x;  
 string y;  
 **int** offset; //地址偏移或立即数  
};  
  
**struct** CodeTable {  
 **struct** Code codeArray[MAXCODE];  
 **int** i; //索引变量  
};

### 寄存器状态

**struct** Reg {  
 string name; //存储的临时变量、局部变量的名字  
 **int** kind; //0-临时数字, 1-临时变量, 2-局部变量, 3-全局变量  
 **int** offAddr; //地址偏移  
 **int** isEmpty; //当前寄存器是否可用  
 **int** isBusy; //之后根据冲突设置当前指令是否使用  
};  
  
**struct** RegTable {  
 **struct** Reg regArray[REGNUM]; //临时寄存器 $8-$25  
 **int** ptr; //指向当前空闲位置或者当前该被替换的寄存器(FIFO)  
 **bool** isFull; //表示是否满了  
};

### 变量状态

**struct** varInfo {  
 **bool** inReg; //是否在寄存器中, 1表示是, 0表示不是  
 **int** regAddr; //寄存器的数组下标 0 - 17  
 **int** stackAddr; //数据栈中相对基地址偏移  
 **bool** isFirst; //临时变量才会用到, 是否为第一次使用  
};

### 其他表

map<string, varInfo> tempMap; 临时变量信息

map<string, varInfo> localMap; 局部变量信息

map<string, int> localCountMap; 局部变量计数

### 关键算法

a. 处理操作数

在处理z、x、y操作数时首先判断其属于数字、临时变量还是全局变量。若为数字，将其返回，暂时不给其分配寄存器。若为临时变量，从tempMap表中查找其所在位置，若在寄存器中则返回相应的寄存器编号，若在内存中则从内存中取出并为其分配一个寄存器并将寄存器号返回。若为局部变量，从localMap中查找其状态，若查不到说明是第一次引用，则通过查符号表得到其在内存中的地址并将信息加入localMap中，然后返回分配的寄存器编号。

b. 寄存器清空策略

i. 清空$8-$25号寄存器，结果不存入内存，函数声明时使用。

ii. 清空$8-$25寄存器, 将相应内容存入内存中，在jal之前使用(保存现场)。

iii. 清空$8-$25寄存器, 将全局变量存入内存,其余不存入，调用返回时使用。

iv. 清空临时变量寄存器，保留全局寄存器，函数内跨基本块时使用。

3>标签处理

根据跳转标签种类的不同有不同的处理方法。

a.普通的跳转标签：先使用上述iv策略清空寄存器，再设置标签。

b.main函数入口标签：设置$fp(函数基址)和$sp(栈顶)的值，分配全局寄存器。

c.函数返回地址标签：恢复栈顶空间$sp，恢复上层函数基址$fp，更新$ra。

d.函数开始标签：打印标签，分配全局寄存器。

## 优化方案\*

### 常量替换

优化连续出现的常量表达式，如a=b+1+2+3,可优化为a=b+6。

### （2）基本块内部公共子表达式删除

**struct** Node {  
 std::string name; //节点所对应的变量名, 中间节点对应属性  
 **int** op\_id; //+,-,\*,/,= 的操作符对应的编号, -1代表其为叶子节点  
 **struct** tree\_node\* lchild; //左孩子  
 **struct** tree\_node\* rchild; //右孩子  
 vector<std::string> varNames; //该节点所对应的变量名称链表  
 **bool** isMid; //是否为中间节点  
 std::vector<**struct** tree\_node\*> parents; //父节点, 在resultProcess()中对两个孩子节点的父节点进行更新操作  
};

首先根据函数的开始标签地址和结束标签地址划分函数块，然后再函数块内找到连续的表达式(加减乘除和赋值语句)进行处理，先根据表达式生成DAG图，再使用启发式算法进行导出。

### （3）寄存器分配、调度优化

1. 利用引用计数分配全局寄存器
2. 利用FIFO策略分配临时寄存器

## 出错处理

|  |  |  |
| --- | --- | --- |
| 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 |
| ID\_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\_ID | 8 | 数字开头的标识符 |
| NUM\_TOO\_LARGE | 9 | 数字太大 |
| ILLEGAL\_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. 跳到下一个双引号位置
3. 跳到下一个逗号位置
4. 跳到下一个括号位置
5. 跳到某一元素集合的位置
6. 跳到下一个类型标识符或void的位置
7. 跳到下一语句列

8、程序直接退出

# 三．操作说明

## 1．运行环境

【说明搭建运行环境的步骤】

## 2．操作步骤

【详细说明操作步骤】

# 四．测试报告

## 1．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】

## 2．测试结果分析

【说明上述测试程序对语法成分的覆盖情况】

# 五．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。