《编译技术》课程设计

文 档

学号：\_\_\_\_\_\_16061092\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_\_张璐\_\_\_\_\_\_\_\_\_

2019年 1 月 4 日

## 一、需求说明

### 1、文法说明

#### 1.1文法内容

获取到的文法为C0难度文法，具体文法如下：

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜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＞ ::= ＜数字＞｛＜数字＞｝

＜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＞‘}’|＜声明头部＞‘{’＜复合语句＞‘}’ //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数定义＞ ::= void＜标识符＞(’＜参数表＞‘)’‘{’＜复合语句＞‘}’| void＜标识符＞{’＜复合语句＞‘}’//第一种选择为有参数的情况，第二种选择为无参数的情况

＜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＞}

＜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＞‘]’|‘(’＜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＞;|＜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＞ ::= while ‘(’＜条件＞‘)’＜语句＞ |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＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

＜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类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符不区分大小写字母

（3）写语句中，字符串原样输出，单个字符类型的变量或常量输出字符，其他表达式按整型输出

（4）数组的下标从0开始

（5）for语句先执行一次循环体中的语句再进行循环变量是否越界的测试

#### 1.2文法解读

##### 1）＜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＞

###### 1.1作用

<程序>是扩充C0文法识别的开始符号，是整个源程序文件的文法格式。

###### 1.2限定条件

1. <程序>可以有常量说明，也可以没有。
2. 之后可以有变量说明，也可以没有。
3. 之后可以有0或多个<有返回值函数定义>或<无返回值函数定义>。
4. 之后必须要有<主函数>。且主函数结束后，程序结束。

###### 1.3句子示例

const int a=0;

const int b=0;

int c;

int d;

int fun1(int fa, char fb) {<复合语句1>}

void fun2 {<复合语句2>}

void main() {<复合语句3>}

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

###### 2.1作用

用于开始常量的定义。

###### 2.2限定条件

1. 最开始必须要有const关键字。
2. 之后必须要有<常量定义>，后面紧跟着一个’;’。
3. 之后还可以有0或多个常量定义。

###### 2.3句子示例

const int fat = 0;

const char thin=1;

##### 3）＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

###### 3.1定义

用于开始变量的定义。

###### 3.2限定条件

1. 必须要有<变量定义>，后面紧跟着一个’;’。
2. 之后还可以有0或多个<变量定义>。

###### 3.3句子示例

int short, big;

char long, curly[123];

##### 4） ＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’|＜声明头部＞‘{’＜复合语句＞‘}’ //第一种选择为有参数的情况， 第二种选择为无参数的情况

###### 4.1作用

用于定义有返回值函数，可以是有参数的有返回值函数，也可以是无参数的有返回值函数。

1.有参数的有返回值函数：＜声明头部＞‘(’＜参数表＞‘)’‘{’＜复合语句‘}’

先是<声明头部>用于说明返回值的类型和参数名。

后面小括号里是<参数表>，表示传入的参数。

接着是{}里面的<复合语句>，是函数的主体部分。

2.无参数的有返回值函数：＜声明头部＞‘{’＜复合语句＞‘}’

<声明头部>用于说明函数的返回值的类型和函数名。

接着是{}里面的<复合语句>，是函数的主体部分。

###### 4.2限定条件

函数的<声明头部>一定要有。{}一定要有。

###### 4.3句子示例

1. int main (<参数表>) {<复合语句>}

2. char sleep{<复合语句>}

##### 5）＜无返回值函数定义＞ ::= void＜标识符＞(’＜参数表＞‘)’‘{’＜复合语句＞‘}’| void＜标识符＞{’＜复合语句＞‘}’ //第一种选择为有参数的情况， 第二种选择为无参数的情况

###### 5.1作用

用于定义无返回值函数，可以是有参数的无返回值函数，也可以是无参数的无返回值函数。

1.有参数的无返回值函数：void <标识符>‘(’＜参数表＞‘)’‘{’＜复合语句‘}’

先是void关键字代表无返回值，<标识符>是函数名。

后面小括号里是<参数表>，表示传入的参数。

接着是{}里面的<复合语句>，是函数的主体部分。

2.无参数的有返回值函数：＜声明头部＞‘{’＜复合语句＞‘}’

先是void关键字代表无返回值，<标识符>是函数名。

接着是{}里面的<复合语句>，是函数的主体部分。

###### 5.2限定条件

void 和<标识符>一定要有。{}也一定要有。

###### 5.3句子示例

1. void main(<参数表>) {<复合语句>}

2. void main{<复合语句>}

##### 6）＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

###### 6.1作用

主函数就是一个特殊的，每个程序只有一个的函数，在常量说明，变量说明和其他函数说明之后出现。主函数结束后程序结束。

###### 6.2限定条件

1. void main()关键字必须要有，不能改变。

2.之后是{}括起来的<复合语句>部分。

3.主函数出现在程序的最后，主函数结束，程序结束。

###### 6.3句子示例

void main() {<复合语句>}

##### 7）＜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＞}

###### 7.1作用

用于定义常量。

###### 7.2限定条件

1.对于int型常量：先是int关键字，然后是<标识符>=<整数>，之后还可以有0或多个<标识符>=<整数>的int型常量的定义。

2.对于char型变量：先是char关键字，然后是<标识符>=<字符>，之后还可以有0或多个<标识符>=<字符>的char型常量的定义。

3.常量定义可以是int型常量的定义，也可以是char型常量的定义

###### 7.3句子示例

1.int fat = 0

2.char long = s

##### 8）＜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

###### 8.1作用

用于定义变量，可用来定义数组。

###### 8.2限定条件

1. 首先是<类型标识符>，<类型标识符> := int | char，即可以定义int型或char型的变量或数组。

2.然后是<标识符>或者<标识符> ‘[’<无符号整数>‘]’。前者表明定义的是变量，变量名为<标识符>，后者表明定义的是数组，数组名为<标识符>。

3.后面可以有0个或多个的以’,’开头的，<类型标识符>类型的，变量或数组的定义。

###### 8.3句子示例

1.int fat, thin, big

2.char long, short[250]

##### 9） ＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

###### 9.1作用

定义<声明头部>，可以是int型的声明，也可以是char型的声明。

###### 9.2限定条件

1.int型的声明：int关键字后面跟着<标识符>，<标识符>就是函数名。

2.char型的声明：char关键字后面跟着<标识符>，<标识符>就是函数名。

###### 9.3句子示例

1. int boring

2. char happy

##### 10）＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

###### 10.1作用

用于有参数的函数定义。

###### 10.2限定条件

1.每个<标识符>变量定义时，必须要带有类型标识符。

2.不同变量定义之间用逗号隔开。

###### 10.3句子示例

int a, int b, char c

##### 11） ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

###### 11.1作用

<复合语句>用于定义函数的主体部分内容。

###### 11.2限定条件

1.<常量说明>可有可无。

2.<变量说明>可有可无。

3.1和2一定是按照顺序来的，不能先<变量说明>再<常量说明>。

###### 11.3句子示例

const int a = 0;

char b;

<语句列>

##### 12）＜语句列＞ ::= ｛＜语句＞｝

###### 12.1作用

用于定义一个或一系列<句子>的组合，也可以是空。

###### 12.2限定条件

无

###### 12.3句子示例

1. //空

2. <语句>

3. <语句1><语句2>

##### 13） ＜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＞;

###### 13.1作用

可以是条件语句，循环语句，大括号括起来的<语句列>，可以调用函数，可以给变量或数组赋值，可以读入，打印，可以是返回语句，也可以是空。

###### 13.2限定条件

可以用来实现某一功能，在函数的<常量说明>和<变量说明>之后。也可以是循环体中的<语句>。

###### 13.3句子示例

1. //空

2.{<语句1><语句2><语句3>}

3.其他部分相见具体的特殊语句的句子示例。

##### 14. ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

###### 14.1作用

用于定义<标识符>，可由<字母>，<数字>组成。

###### 14.2限定条件

1.首先一定要有一个<字母>。

2.接着后面可以跟0或多个<字母>|<数字>。

###### 14.3句子示例

1. \_

2. a

3. \_aa

4. ss2a

##### 15） ＜类型标识符＞ ::= int | char

###### 15.1作用

<类型标识符>的定义，只能是int类型或char类型。

###### 15.2限定条件

只能是int或char中的一个

###### 15.3句子示例

1. int

2. char

##### 16） ＜条件语句＞::= if ‘(’＜条件＞‘)’＜语句＞[else＜语句＞]

###### 16.1作用

1.满足条件的执行if后面跟的<语句>。

2.不满足<条件>的，若后面还有else，执行else后的<语句>。

3.不满足<条件>的，若后面没有else，<条件语句>结束。

###### 16.2限定条件

1.<条件>里的各<表达式>的值已知。

2.If后面的else<语句>可有可无

###### 16.3句子示例

1. if a>=b <语句1> else <语句2> //a，b值已知

2. if a>=c <语句> //a，c值已知

##### 17）＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞ |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞

###### 17.1作用

1.while循环语句：满足<条件>的，执行while后的<语句>，不满足条件，<循环语句>结束。

2.for循环语句：先初始化＜标识符＞＝＜表达式＞，再执行一次for之后的<语句>，再执行＜标识符＞＝＜标识符＞(+|-)＜步长＞，再判断是否满足<条件>，满足条件，继续执行<语句>，执行完先执行＜标识符＞＝＜标识符＞(+|-)＜步长＞，再判断<条件>，循环往复，直到<条件>不满足，<循环语句>结束。

###### 17.2限定条件

<条件>里各<表达式>的值已知。

###### 17.3句子示例

1.while(a>=b) <语句> //当需要判断<条件>时，a和b的值已知。

2.for( i=1;i<100;i=i+1) {i=i+1;}

##### 18） ＜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

###### 18.1作用

1.用于条件语句或循环语句中的条件判断。

2.可以是表达式之间关系判断，如果表达式间的关系满足关系运算符限定关系，则<条件>为真，否则条件为假。

3.也可以是单独的表达式，若表达式值为0，条件为假，否则为真。

###### 18.2限定条件

各<表达式>里各<项>的值已知。

###### 18.3句子示例

1.a ==b (a,b的值已知)

2. a (a的值已知)

##### 19） ＜步长＞::= ＜无符号整数＞

###### 19.1作用

用于循环语句中for语句每执行完一次循环体<语句>，<标识符>的自增或自减。

###### 19.2限定条件

无

###### 19.3句子示例

1

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

###### 20.1作用

为变量或数组赋值。

###### 20.2限定条件

赋值时，等号右边的表达式中各项的值已知。

###### 20.3句子示例

1. a = b //b的值已知

2. a[n] = b // n, b值已知

##### 21） ＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况， 第二种选择为无参数的情况

###### 21.1作用

调用有返回值的函数，可以作为因子，也可以作为语句。

###### 21.2限定条件

1.<标识符>用于说明调用的函数名，必须要有。

2.对于有参数的函数，<值参数表>中的参数类型必须符合函数的参数表的参数类型。

3.<值参数表>中的<表达式>的各<项>的值必须已知。

###### 21.3句子示例

1. fat(a, b, c) //已定义有参数有返回值的函数fat，且传入的a，b，c类型与定义函数时的参数类型一致。

2. thin //已定义无参数有返回值的函数thin。

##### 22） ＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况， 第二种选择为无参数的情况

###### 22.1作用

用于调用无返回值的函数，不能作为因子，只能作为语句。

###### 22.2限定条件

1.<标识符>用于说明函数名，必须要有。

2. 对于有参数的函数，<值参数表>中的参数类型必须符合函数的参数表的参数类型。

3.<值参数表>中的<表达式>的各<项>的值必须已知。

###### 22.3句子示例

1. fat(a, b, c) //已定义有参数有返回值的函数fat，且传入的a，b，c类型与定义函数时的参数类型一致。

2. thin //已定义无参数有返回值的函数thin。

##### 23）＜表达式＞ ::= ［＋｜－］ ＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

###### 23.1作用

用于定义表达式，可以描述很多项相加，也可以是单项。

###### 23.2限定条件

1.若是单项，项前可以是+，也可以是-，也可以什么都没有。

2.若不是单项，若第一个项前面有符号，第一个项前面的+或-只作用于第一个项。

3.若不是单项，第一个项后面可以有1或多个(<加法运算符><项>)。

###### 23.3句子示例

1. 5\*n

2. +a-b+(c\*d)

##### 24） ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

###### 24.1作用

用于定义<项>，可以描述多个因子想乘，也可以只有单独一个因子。

###### 24.2限定条件

1.首先，<因子>一定要有。

2.后面可以跟0或多个（<乘法运算符><因子>）。

###### 24.3句子示例

a[n]\*b[5]/45

##### 25) ＜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＞

###### 25.1作用

用于定义<因子>，由<标识符>，]，<表达式>，]，(，)，<整数>，<字符>，<有返回值函数调用语句>中的一个或一些组成。

###### 25.2限定条件

1.<因子>可以单独由<标识符>组成。示例1

2.<因子>可以由＜标识符＞‘[’＜表达式＞‘]’组成。示例2

3.<因子>可以由‘(’＜表达式＞‘)’组成。示例3

4.<因子>可以由＜整数＞单独组成。示例4

5.<因子>可以由<字符>单独组成。示例5

6.<因子>可以由<有返回值函数调用语句>单独组成。

7. 若<标识符>类型为char类型的变量或常量，用字符的 ASCII 码对应的整数参加运算。

###### 25.3句子示例

1. \_abc

2. \_abc[n]

3. (a+b)

4. 23

5. ‘a’

##### 26) ＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

###### 26.1作用

用于有参数的函数调用语句传入参数。

###### 26.2限定条件

<表达式>中的各<项>的值必须已知。

###### 26.3句子示例

a, b, c + d, e-f

##### 27) ＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

###### 27.1作用

用于读入一个或多个名字为<标识符>的变量。

###### 27.2限定条件

1. scanf关键字

2.后面要读入的一个或多个<标识符>的最外层要有小括号。

3.<标识符>只能是变量名。

###### 27.3句子示例

1. scanf(a) //a为已定义的变量名

2. scanf(a,b) //a,b均为已定义的变量名。

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

###### 28.1作用

用于打印字符串或表达式。

###### 28.2限定条件

1. printf关键字一定要有。

2. 要打印的所有的<字符串>或<表达式>的最外层一定要有小括号。

###### 28.3句子示例

1. printf(“aisubfiuid”)

2. printf(“asda”,<表达式>)

3. printf(<表达式>)

##### 29)＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

###### 29.1作用

用于定义<返回语句>，返回一个表达式。

###### 29.2限定条件

1.return关键字一定要有。

2.表达式可有可无，有的话<表达式>外部需要嵌套上‘(’， ‘)’。

###### 29.3句子示例

1. return

2. return ‘a+b+c’

##### 30) ＜整数＞ ::= ［＋｜－］＜无符号整数＞

###### 30.1作用

定义<整数>，由符号和无符号整数组成。

###### 30.2限定条件

<无符号整数>前面可以是+，可以是-，可以什么都没有

###### 30.3句子示例

1. 23

2. +23

3. -23

##### 31) ＜无符号整数＞ ::= ＜数字＞｛＜数字＞｝

###### 31.1作用

定义<无符号整数>，由数字组成。

###### 31.2限定条件

由一个或多个<数字>组成

###### 31.3句子示例

1. 2

2. 23

3. 3232

##### 32)＜数字＞ ::= ０｜１｜．．．｜９

###### 32.1作用

定义<数字>

###### 32.2限定条件

只能是0-9整数中的一个

###### 32.3句子示例

2

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

###### 33.1作用

用于定义<字符>。

###### 33.2限定条件

字符外边被' '包住。

###### 33.3句子示例

1. 'a'

2. '3'

3. '+'

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

###### 34.1作用

规定字符串中可以有的字符

###### 34.2限定条件

1.规定字符串中可以有的字符为ASCII码值在十进制32，33，35-126之间的字符。

2.字符串外边用双引号括起来

###### 34.3句子示例

“aaa+23”

##### 35) ＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

###### 35.1作用

给出<字母>的定义

###### 35.2限定条件

只能是\_或26个英文字母的大小写中的一个

###### 35.3句子示例

a

##### 36) ＜加法运算符＞ ::= +｜-

###### 34.1作用

给出<加法运算符>的定义

###### 34.2限定条件

只能是+或-中的一个

###### 34.3句子示例

+

##### 37) ＜乘法运算符＞ ::= \*｜/

###### 34.1作用

给出<乘法运算符>的定义

###### 34.2限定条件

只能是\*或/中的一个

###### 34.3句子示例

\*

##### 38)＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

###### 34.1作用

给出<关系运算符>的定义

###### 34.2限定条件

只能是<或<=或>或>=或!=或==中的一个

###### 34.3句子示例

<=

#### 1.3改写说明

虽然文法中有几处规则右部FIRST集相交的情况，但是没有对文法进行改写，只是在实现的时候，利用trace\_words()函数回溯完成对不同文法规则的识别。

### 2、目标代码说明

目标代码为mips汇编代码。关于mips汇编代码的说明详细不再赘述，此处只给出老师在编译公告上公开发布的寄存器使用说明和生成目标代码时可能会使用到的mips指令。

寄存器使用说明：

|  |  |  |
| --- | --- | --- |
| REGISTER | NAME | USAGE |
| $0 | $zero | 常量0 |
| $1 | $at | 保留给汇编器 |
| $2-$3 | $v0-$v1 | 函数调用返回值 |
| $4-$7 | $a0-$a3 | 函数调用参数 |
| $8-$15 | $t0-$t7 | 临时寄存器 |
| $16-$23 | $s0-$s7 | 全局寄存器 |
| $24-$25 | $t8-$t9 | 临时寄存器 |
| $28 | $gp | 全局指针(Global Pointer) |
| $29 | $sp | 堆栈指针(Stack Pointer) |
| $30 | $fp | 帧指针(Frame Pointer) |
| $31 | $ra | 返回地址(return address) |

可能用到的mips汇编指令：

|  |  |
| --- | --- |
| 目标指令 | 含义 |
| add res op1 op2 | res = op1 + op2 |
| sub res op1 op2 | res = op1 – op2 |
| mult op1 op2  mflo res | res = op1 \* op2 |
| div op1 op2  mflo res | res = op1 / op2 |
| sw op1,op2(res) | res[op2] = op1 |
| lw res,op2(op1) | res = op1[op2] |
| J op1 | GOTO op1 |
| Jal op1 | 对于无返回值函数调用  GOTO op1  返回地址存在$ra  //op为GOTO,op1存label |
| Jal op1  sw $t0 (res) | 对于有返回值函数调用  GOTO op1  返回地址存在$ra  //op为GOTO,op1存label，res存中间变量，此中间变量存储返回值$t0 |
| sub t1 x y  bgez t1 label | x<y  BZ label  //op为<  //op1存x  //op2存y |
| sub t1 x y  bgtz t1 label | x<=y  BZ label  //op为<=  //op1存x  //op2存y |
| sub t1 x y  blez t1 label | x>y  BZ label  //op为>  //op1存x  //op2存y |
| sub t1 x y  bltz t1 label | x>=y  BZ label  //op为>=  //op1存x  //op2存y |
| beq x y label | x!=y  BZ label  //op为!=  //op1存x  //op2存y |
| op1：  表示标签 | label op1： |
| bne x y label | x==y  //op为==  //op1存x  //op2存y  //res存真假 |
| li $v0 12  syscall | 读入char型 |
| li $v0 5  syscall | 读入int型 |
| li $a0 adr  li $v0 4  syscall | 打印字符串,字符串地址为adr |
| 将integer装入 $a0  li $v0 1  syscall | 打印整型integer |
| 将character装入$a0  li $v0 11  syscall | 打印字符型character |
| li $v0 10  syscall | Exit |

### 3. 优化方案

**3.1利用基本块划分算法划分基本块**。

• 输入： 四元式序列

• 输出： 基本块列表， 每个四元式仅出现在一个基本块中

• 方法：

– 1、 首先确定入口语句（每个基本块的第一条语句） 的集合

• 规则1： 整个语句序列的第一条语句属于入口语句

• 规则2： 任何能由条件 /无条件跳转语句转移到的第一条

语句属于入口语句

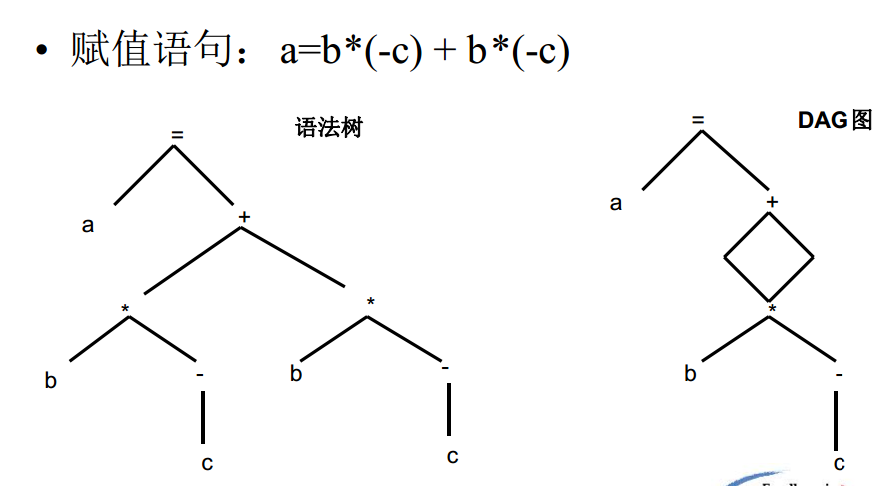
• 规则3： 紧跟在跳转语句之后的第一条语句属于入口语句

– 2、 每个入口语句直到下一个入口语句， 或者程序结束， 它

们之间的所有语句都属于同一个基本块

**3.2 基本块内用DAG图消除局部公共子表达式**。

DAG图：（此图来源于老师课件）



得到DAG图后，可以利用从DAG图导出中间代码的启发式算法：

• 输入： DAG图

• 输出： 中间代码序列

• 方法：

1. 初始化一个放置 DAG图中间结点的队列。

2. 如果DAG图中还有中间节点未进入队列， 则执行步骤 3， 否则执行步骤 5

3. 选取一个尚未进入队列， 但其 所有父节点均已进入队列 的中间节点 n，

将其加入队列； 或选取 没有父节点 的中间节点， 将其加入队列

4. 如果n的最左子节点 符合步骤3的条件， 将其加入队列； 并沿着当前节点

的最左边， 循环访问其最左子节点 ， 最左子节点的最左子节点等， 将符

合步骤3条件的中间节点依次加入队列； 如果出现不符合步骤 3条件的最

左子节点， 执行步骤 2

5. 将中间节点队列 逆序输出， 便得到中间节点的计算顺序， 将其整理成中

间代码序列

**3.3 全局优化**

**3.3.1.利用活跃变量数据流分析构建冲突图**。

活跃变量数据流方程如下：

– in[B] = use[B] ∪ (out[B] – def[B])

– out[B] = ∪B的后继基本块 P in[P]

– def[B] ： 变量在B中被定义（赋值） 先于任何对它们的使用

– use[B] ： 变量在B中被使用先于任何对它们的定义

利用冲突图可找到可以共享寄存器的变量。

**3.3.2 全局寄存器分派问题：**

全局寄存器的图着色算法：

在利用活跃变量数据流分析得到冲突图后，我们可以基此算法进行全局寄存器分配。

输入：变量冲突图G，全局寄存器数量K，对应K中不同颜色。

输出：全局寄存器分配结果，变量到寄存器的对应关系。

**3.3.3利用到达定义数据流分析构建定义-使用链**。

到达定义数据流信息：

–out[B] = gen[B] ∪ ( in[B] – kill[B] )

– in[B]为进入基本块时的数据流信息

– kill[B] = kill[d1] U kill[d2]…U kill[dn]， d1~dn依次为基本块中的语句

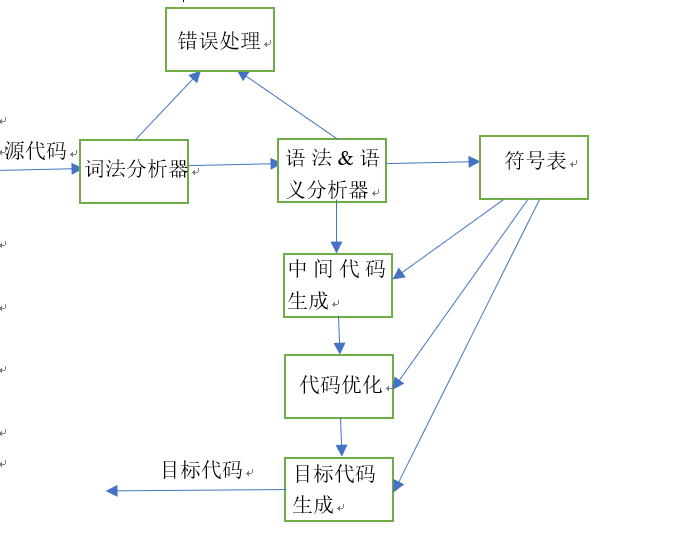
– gen[B] = gen[dn] U(gen[d(n-1)] – kill[dn]) U(gen[d(n-2)] – kill[d(n-1)] – kill[dn])… U(gen[d1] - kill[d2] – kill[d3]…- kill[dn])

将不同变量的不同定义点（如a），将该定义点与所有可能用到该点定义的a的使用点连成定义-使用链。

将得到的链合并，得到网。根据由网构成的冲突图来完成寄存器分配后，生成代码时，将根据当前变量属于哪个网来决定选取哪个寄存器代替该变量。

## 二、详细设计

### 1．程序结构



### 2．类/方法/函数功能

**2.1词法分析函数(\_getsym)**

分析得到每个词的符号类型symcon，获取到的具体符号content，以及这个符号的行号line。当输入字符不满足时，调用错误处理函数error()。

函数：

\_getsym()函数用于词法分析，将单词存入char类型的words数组里，并将单词的类型存入int类型的words\_sym数组里。

getsym()函数用于返回单词类型。用于语法分析。

词法分析sym设计图：

|  |  |  |
| --- | --- | --- |
| sym | symcon | content |
| 1 | constsy | const |
| 2 | intsy | int |
| 3 | charsy | char |
| 4 | voidsy | void |
| 5 | mainsy | main |
| 6 | ifsy | if |
| 7 | elsesy | else |
| 8 | whilesy | while |
| 9 | forsy | for |
| 10 | scanfsy | scanf |
| 11 | printfsy | printf |
| 12 | returnsy | return |
| 13 | identcon | (见具体) |
| 14 | charcon | (见具体) |
| 15 | stringcon | (见具体) |
| 16 | us\_intcon | (见具体) |
| 17 | plus | + |
| 18 | minus | - |
| 19 | times | \* |
| 20 | div | / |
| 21 | less\_sy | < |
| 22 | less\_e\_sy | <= |
| 23 | more\_sy | > |
| 24 | more\_e\_sy | >= |
| 25 | not\_e\_sy | != |
| 26 | givez\_sy | = |
| 27 | e\_sy | == |
| 28 | semicolon | ; |
| 29 | lparent | ( |
| 30 | rparent | ) |
| 31 | lbrack | [ |
| 32 | rbrack | ] |
| 33 | lbrace | { |
| 34 | rbrace | } |
| 35 | comma | , |

**2.2语法分析&语义分析器&中间代码生成调用函数 (grammar)**

**基本功能：**

初始化一系列行列信息，符号表信息，预读一个单词后，进入procedure()函数，开始语法分析&语义分析&中间代码生成。

会调用一系列函数，基于词法分析得到的单词，利用递归下降分析法，按照文法对源代码进行语法分析，若有不符合文法要求的，需调用出错处理函数error()。之后对某些语法成分进行语义分析，对于不符合语义要求的调用出错处理函数error()。最后，对于满足语法和语义要求的成分，直接生成中间代码。

Procedure()函数返回后，源代码分析结束，若过程中间没有语法和语义错误，此时已经生成中间代码。

最后是检查是否有“有返回值函数是否有return一个返回值”的语义错误，若有此语义错误，直接报错，并指明是哪个函数没有返回值。

调用到的语法分析函数有：

void procedure();

void const\_dec();

void const\_def();

void var\_dec();

void var\_def();

void res\_fuc();

void nores\_fuc();

void param();

char\* factor();

char\* item();

char\* exprs();

void val\_param();

void com\_stam();

void stam\_q();

int stam();

void if\_stam();

void circle\_stam();

void condit();

void res\_call(int fuc\_i);

void nores\_call(int n\_fuc\_i);

void ass\_stam();

void read\_stam();

void write\_stam();

void return\_stam(int ret\_i);

void main\_fuc();

下面分别对其进行解释。

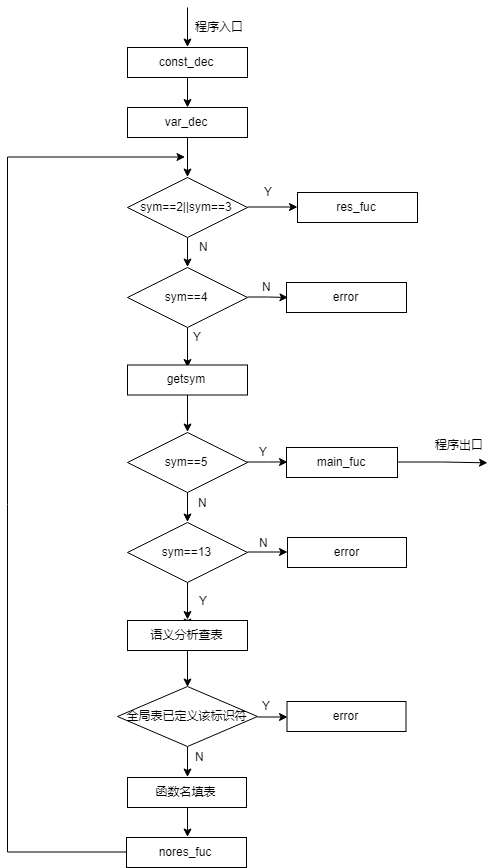
###### void procedure()

**基本功能：**

主要实现对以下文法规则的识别：

＜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＞

**算法框图：**

****

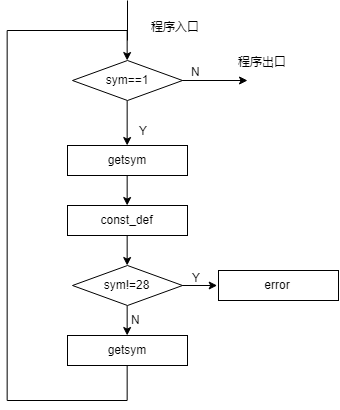
###### void const\_dec()

**基本功能：**

主要实现对以下文法规则的识别：

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

**算法框图：**



###### void const\_def()

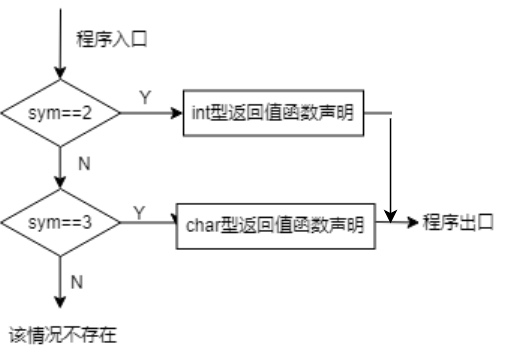
**基本功能：**

主要实现对以下文法规则的识别：

＜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＞}

**关键算法框图：**



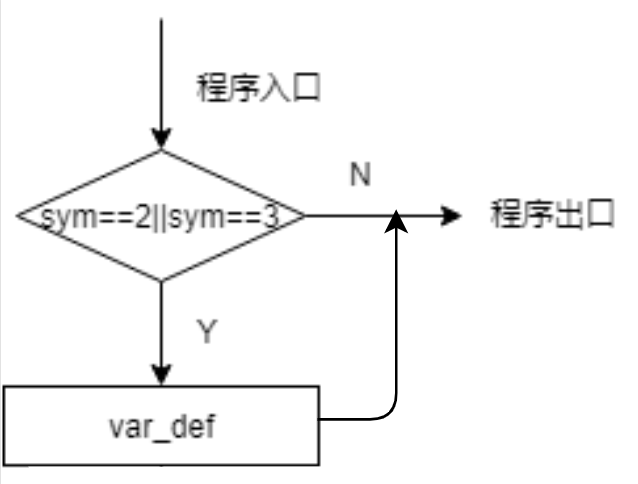
###### void var\_dec()

**基本功能：**

主要实现对以下文法规则的识别：

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

**关键算法框图：**



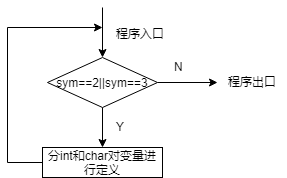
###### void var\_def()

**基本功能：**

主要实现对以下文法规则的识别：

＜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

**关键算法框图：**

****

因此规则与<有返回值函数定义>的FIRST集相交，因此在进入“分int和char对变量进行定义”中，需要判断文法满足的是哪个规则，若是<有返回值函数定义>，需要回溯。

**＜无符号整数＞表示数组元素的个数，其值需大于0**的实现：

该条件属于语义分析部分需要满足的条件，识别出表示数组元素个数的<无符号整数>之后，利用atoi()函数判断其是否满足大于0。不满足调用error()，满足则将数组大小填入符号表中，并生成相应的中间代码。

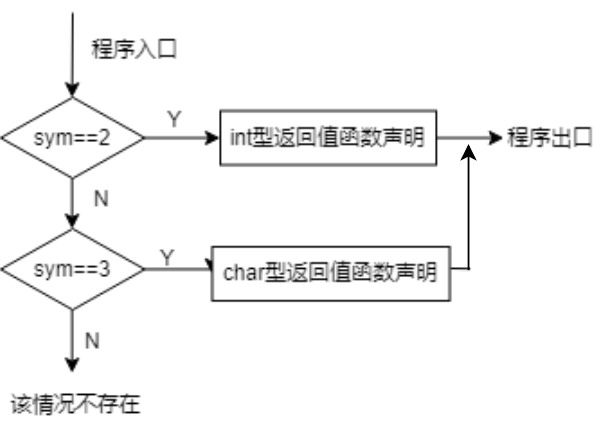
###### void res\_fuc()

**基本功能：**

主要实现对以下文法规则的识别：

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’|＜声明头部＞‘{’＜复合语句＞‘}’ //第一种选择为有参数的情况，第二种选择为无参数的情况

**关键算法框图：**



在具体函数声明部分，每个声明都需要查全局表，若有重名则调用error()，否则将函数名填入全局表，同时创建一个新的函数表。

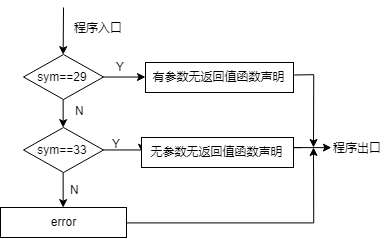
###### void nores\_fuc()

**基本功能：**

主要实现对以下文法规则的识别：

＜无返回值函数定义＞ ::= void＜标识符＞(’＜参数表＞‘)’‘{’＜复合语句＞‘}’| void＜标识符＞{’＜复合语句＞‘}’//第一种选择为有参数的情况，第二种选择为无参数的情况

**关键算法框图：**

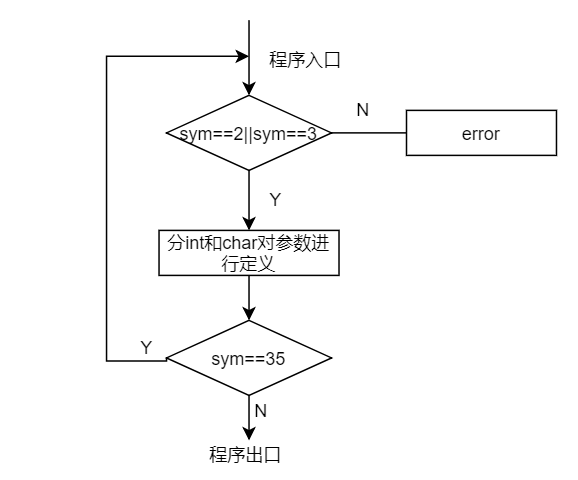
****

###### void param()

**基本功能：**

主要实现对以下文法规则的识别：

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}**关键算法框图：**



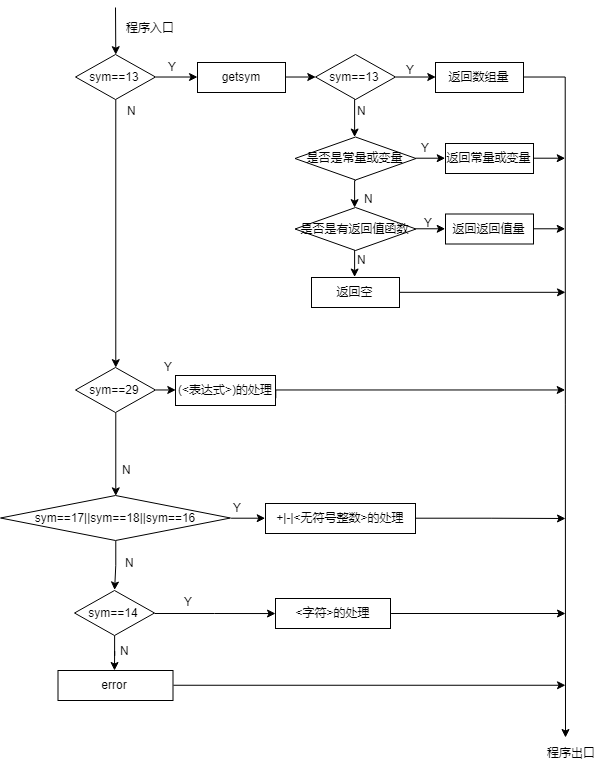
###### char\* factor()

**基本功能：**

主要实现对以下文法规则的识别：

＜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＞

**关键算法框图：**



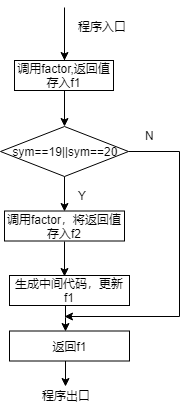
###### char\* item()

**基本功能：**

主要实现对以下文法规则的识别：

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

**关键算法框图：**



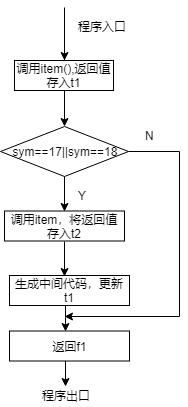
###### char\* exprs()

**基本功能：**

主要实现对以下文法规则的识别：

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

**关键算法框图：**



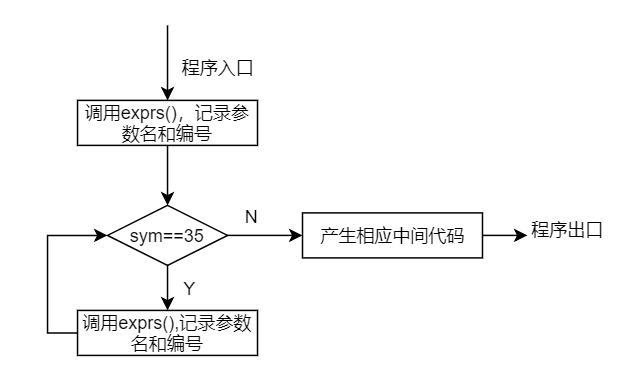
###### void val\_param()

**基本功能：**

主要实现对以下文法规则的识别：

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

**关键算法框图：**



###### void com\_stam()

**基本功能：**

主要实现对以下文法规则的识别：

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

**算法流程：**

程序入口🡪const\_dec()🡪var\_dec()🡪stam\_q()🡪程序出口

###### void stam\_q()

**基本功能：**

主要实现对以下文法规则的识别：

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

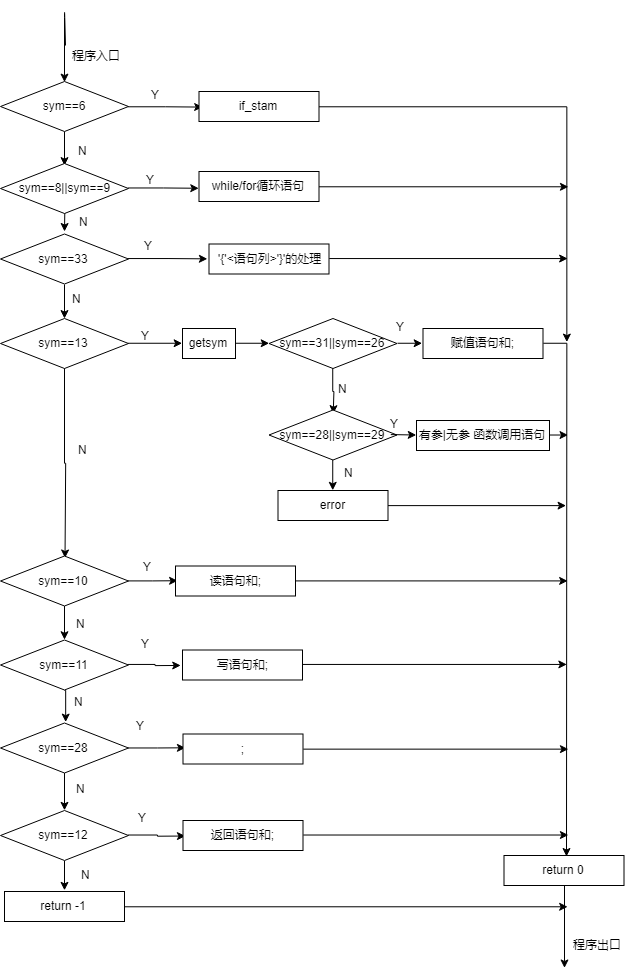
###### int stam()

**基本功能：**

主要实现对以下文法规则的识别：

＜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＞;

**算法框图：**



###### void if\_stam()

**基本功能：**

主要实现对以下文法规则的识别：

＜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

**关键算法：**

针对于形如

if(条件){A}

else{B}

的if语句，中间代码如下：

条件

BZ label\_else

A

GOTO label\_out

label\_else:

B

label\_out:

###### void circle\_stam()

**基本功能：**

主要实现对以下文法规则的识别：

＜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＞ |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＞

**关键算法：**

while:

针对形如

while(<条件>) <语句>

的while语句，中间代码如下：

label\_cir:

条件

BZ label\_out

<语句>

GOTO label\_cir

label\_out:

for:  
针对形如

for(<ident>=<exs>;<condi>;<ident>=<ident>+|-<步长>)<stam>

的for语句，中间代码如下：  
 <ident>=<exs>

label\_cir:

<stam>

<ident>=<ident>+|-<步长>

<condi>

BZ label\_out

GOTO label\_cir

label\_out:

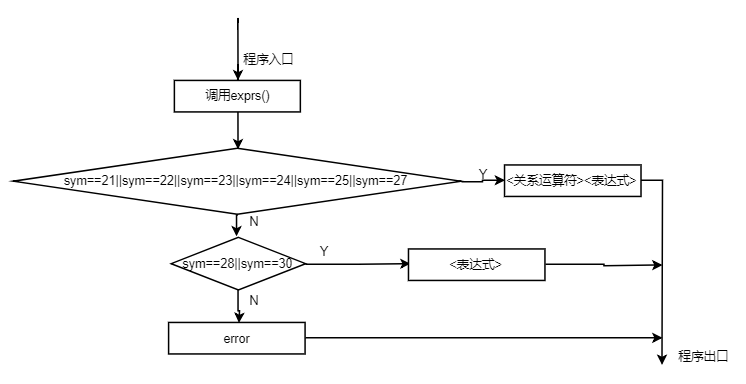
###### void condit()

**基本功能：**

主要实现对以下文法规则的识别：

＜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

**关键算法框图：**



###### void res\_call(int fuc\_i)

**基本功能：**

主要实现对以下文法规则的识别：

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

**关键算法：**

1. 先查表，确定是否有参数，标志量：have\_para。
2. 读入symcon是‘(’，进入val\_para()，对不符合文法的调用error()进行报错处理。
3. 若调用的函数有参数但没有识别到‘(’，调用error()。
4. 完成后产生相应的中间代码。

###### void nores\_call(int n\_fuc\_i)

**基本功能：**

主要实现对以下文法规则的识别：

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

**关键算法：**

此处处理情况与res\_call()类似，不再赘述。

###### void ass\_stam()

**基本功能：**

主要实现对以下文法规则的识别：

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

**关键算法：**

1. 因进入ass\_stam()函数的时候，语法分析当前读入的单词已经在<标识符>的下一个，此处需调用trace\_words(1)回溯一个单词。
2. 分为给数组变量赋值和给非数组变量赋值进行处理。

###### void read\_stam()

**基本功能：**

主要实现对以下文法规则的识别：

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

**关键算法：**

getsym后，当前symcon为‘(’，继续getsym，否则error。

当前symcon为<标识符>，且<标识符>类型不能是const型，否则error。

getsym后，当前symcon是‘，’，则返回上一步，否则继续下一步。

getsym后，当前symcon为‘)’，继续getsym，否则error。

###### void write\_stam()

**基本功能：**

主要实现对以下文法规则的识别：

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

**关键算法：**

分为有字符串输出和无字符串输出进行处理。

###### void return\_stam(int ret\_i)

**基本功能：**

主要实现对以下文法规则的识别：

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

**关键算法：**

分为有返回值的return语句和无返回值的return语句（return）进行处理。

###### void main\_fuc()

**基本功能：**

主要实现对以下文法规则的识别：

＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

**关键算法：**

1. 依次识别void, main, (, ), {, 中间有不符合的需要调用error报错。
2. 进入复合语句com\_stam()。
3. 识别当前symcon是否是’}’，不是则报错。

**2.3符号表(fill\_table())**

**基本功能：**

在语法分析，语义分析之后，有些成分需要填表，此时需要用到fill\_table()函数。

**关键算法：**

按照嵌套层次level的不同进行填表，当level为1时，表明是函数内部的填表，即填入局部符号表。

当level为0时，表明是函数体外部的填表。当标识符为函数名时，不仅要将函数名填入全局符号表，还需要新建一个局部符号表，用来存储该函数的局部符号信息。当标识符不是函数名时，只将其填入全局符号表即可。

**2.4错误处理（error()）**

分类处理词法分析，语法分析，语义分析中遇到的错误并进行报错。详见本章的10中的错误处理。

**2.5代码优化（Opti）**

详见8.优化方案

**2.6目标代码生成（gen\_aim()）**

**基本功能：**

生成mips汇编代码，能够在MARS上运行。

**关键算法：**

**gen\_aim():**将全局符号表中的所有的全局变量和所有的字符串填入.data。

生成j $$main指令(因为我在函数名前面都加了两个$$，避免冲突)。

对于每个函数体，先生成它的label。

再将$sp值赋给$fp，并将返回地址$ra压栈。

算出该函数需要存在运行栈里的变量的所占空间大小size，$sp自减4\*size。

进入函数主体process\_maif();

函数主体处理完之后，针对于非main函数生成一系列返回调用方的指令。针对于main函数，生成程序结束的指令。

**Process\_maif():**

针对于每种中间代码（opcon）生成目标代码。

**int ret\_identyp()：**

根据标识符名查表得到标识符的种类。

**void get\_op1():**

求出第一个操作数，load到$t0

**void get\_op1\_2a0():**

求出第一个操作数，load到$a0

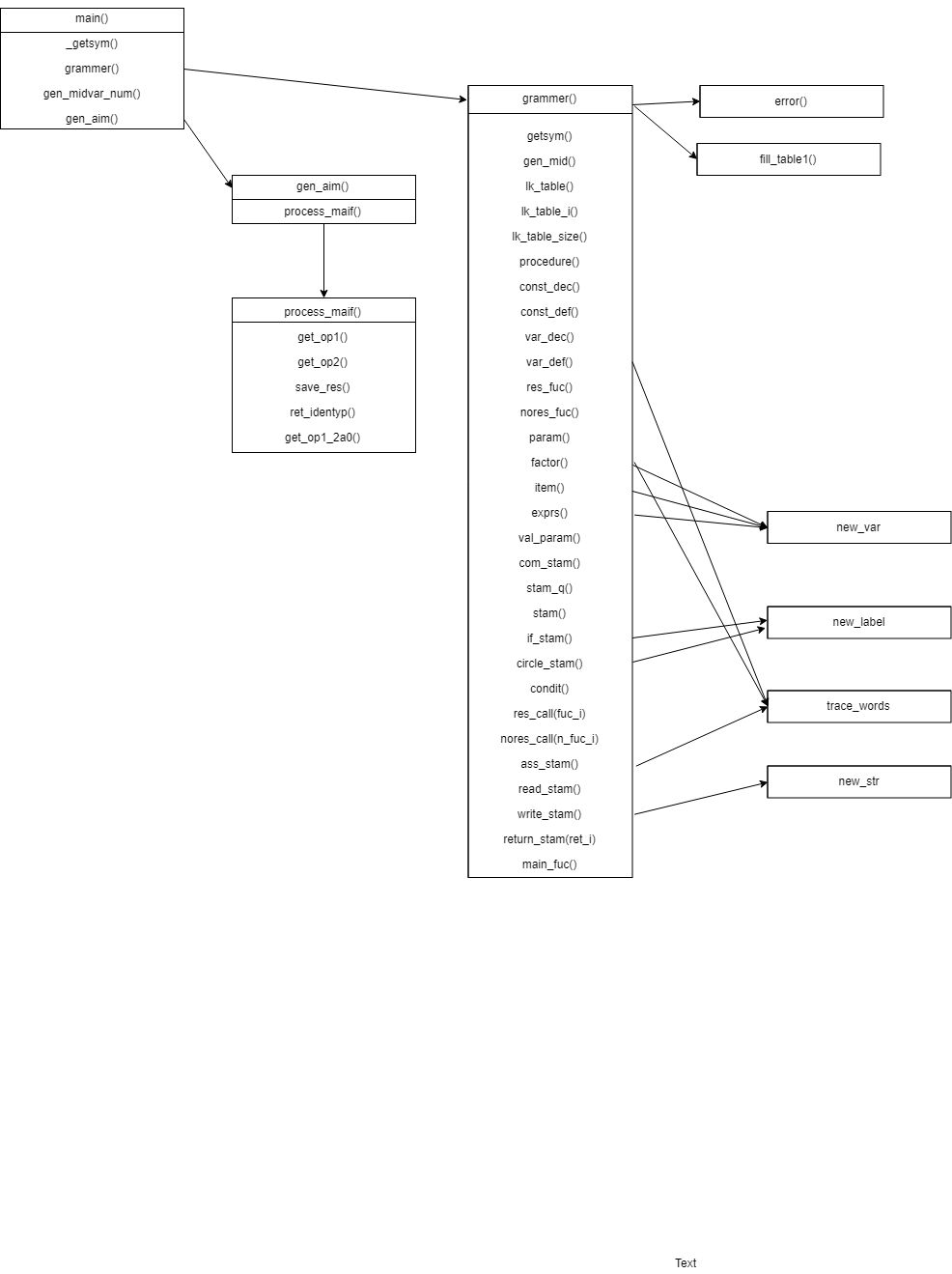
**void get\_op2():**

求出第二个操作数，load到$t0

**void save\_res()：**

此时已默认结果存在$t0里，直接将局部变量存到相应的全局变量int|局部变量int|中间变量|参数。

### 3．调用依赖关系



### 4．符号表管理方案

数据结构：

typedef struct{

char ident\_name[N];

int typ;

int lev;

int adr;

int size;

}Table; //符号表项

typedef struct{

Table table\_contents[MAX\_IDENT\_NUM];

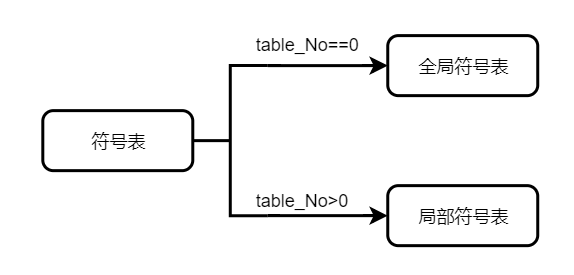
}Tables;///符号表

Tables tables\_local[MAX\_FUC\_NUM];///全体符号表,下标为0时是全局符号表，其余为函数符号表

int table\_No;///符号表当前个数，指向最新生成的符号表

int local\_index[MAX\_FUC\_NUM];///每个函数表的指针，指向下一个要填入的表项

int have\_return[MAX\_FUC\_NUM];///每个函数是否有return语句



|  |  |  |  |
| --- | --- | --- | --- |
| 全局符号表 | Name | 函数符号表 | Name |
| Typ | Typ |
| lev | lev |
| adr | adr |
|  | size |  | size |

其中name为标识符名，typ为标识符种类和类型，lev为该标识符所在分程序的嵌套层数，adr对常数来说，就是它们的值/ASCII码值，对变量来说，是该变量在运行栈S中分配的相对地址，对函数来说，就是他们的目标代码的入口地址。Size是所占空间大小。

其中typ的类型有：

|  |  |
| --- | --- |
| 标识符种类&类型 | typ编号 |
| Char | 1 |
| Int | 2 |
| Const char | 3 |
| Const int | 4 |
| Char[] | 5 |
| Int[] | 6 |
| Res\_char | 7 |
| Res\_int | 8 |
| void | 9 |
| Int型参数 | 10 |
| Char型参数 | 11 |

全局符号表：按照顺序存储全局常量，全局变量，函数名及它们的相关属性。

局部符号表：每个函数有个局部符号表。在每个局部符号表中，按照顺序存储参数，局部常量，局部变量及它们的相关属性。

|  |  |  |  |
| --- | --- | --- | --- |
| Table\_No | Tables\_local | 表内指针 | 当前表项的下一个表项 |
| 0 | 全局符号表 | Local\_index[0] | Tables\_contents[Local\_index[0]] |
| 1 | Fuc1符号表 | Local\_index[1] | Tables\_contents[Local\_index[1]] |
| 2 | Fuc2符号表 | Local\_index[2] | Tables\_contents[Local\_index[2]] |

### 5．存储分配方案

运行时存储组织及管理：

每次调用，旧的$fp压栈，$fp🡨$sp,$ra压栈。

1. 参数区

在调用方就已预留了位置，且已存入栈中。

1. 局部变量区

空间是直接留好的。

1. 中间变量区。
2. 函数返回值。

运行栈结构：

$sp

$fp

地址增长方向

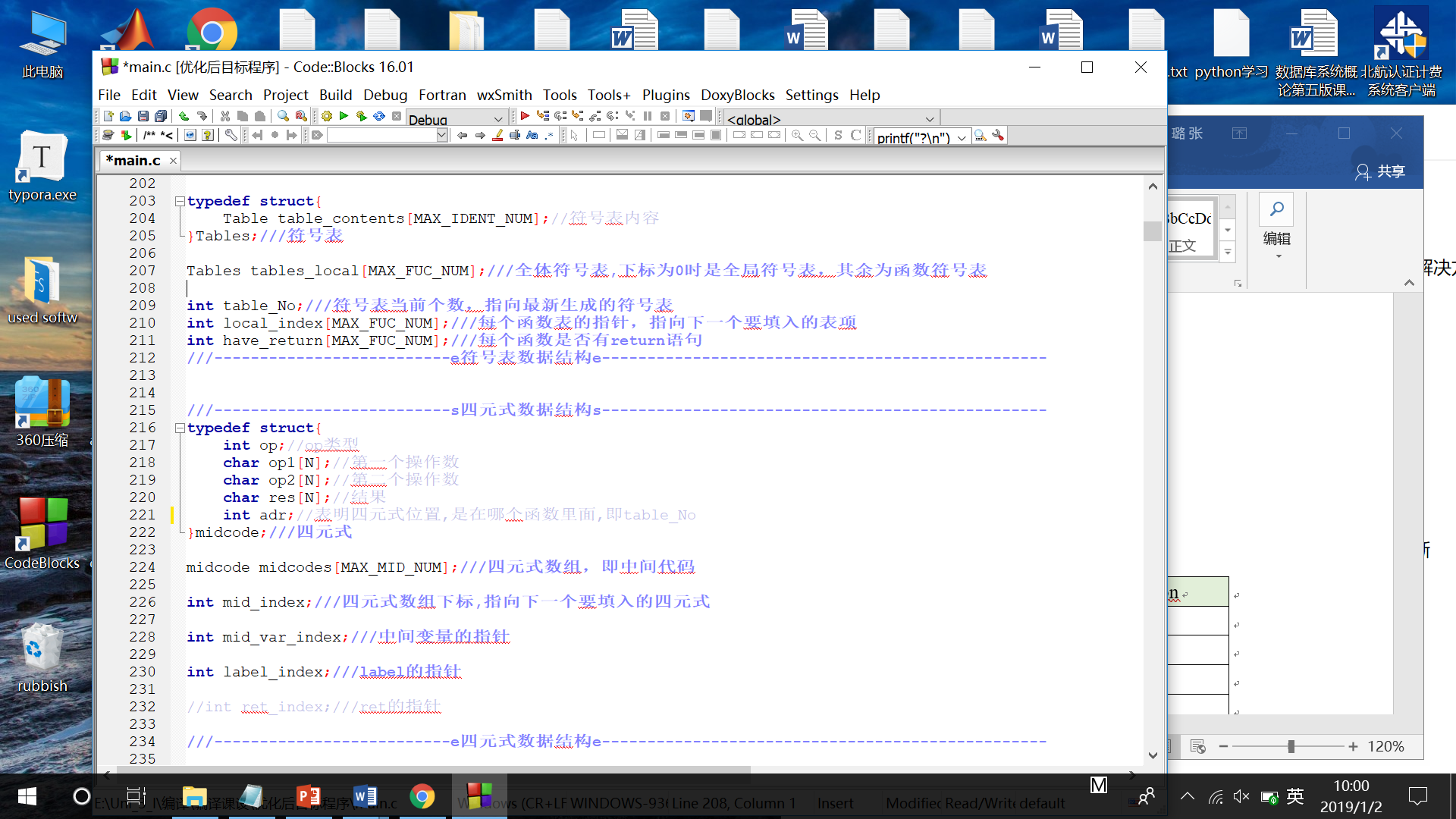
|  |
| --- |
|  |
| $fp’ |
| $ra |
| 参数区 |
| 局部变量区 |
| 中间变量区 |
| 返回值 |
|  |

### 6. 四元式设计

四元式设计格式：

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 第一个操作数 | 第二个操作数 | Result（第三个操作数） |
| OP | op1 | op2 | res |

四元式实际数据结构：



四元式详细设计：

|  |  |  |  |
| --- | --- | --- | --- |
| 源码 | 中间代码 | op | opcon |
| Res = op1 + op2 | Res = op1 + op2 | 1 | + |
| Res = op1 – op2 | Res = op1 – op2 | 2 | - |
| Res = op1 \* op2 | Res = op1 \* op2 | 3 | \* |
| Res = op1 / op2 | Res = op1 / op2 | 4 | / |
| Res[op2] = op1  //为数组中值赋值 | Res[op2] = op1 | 5 | []= |
| Res = Op1[op2]  //为数赋数组中的值 | Res = Op1[op2] | 6 | =[] |
| fuc (x,y) | push x  push y  call fuc  // op1是x,y  //op1是fuc | 7 | push |
| 8 | Call  //无返回值函数调用语句 |
| Res = op1() | call op1  Res = RET | 9 | calr  //有返回值函数调用语句 |
| return x | ret x  //op为ret  //op1为x | 10 | ret |
| Skip to label | GOTO label  //op为GOTO,op1存label | 11 | GOTO |
| If op1<op2, skip to res | x<y  //op为<  //op1存x  //op2存y  //res存真假 | 12 | < |
| If op1<=op2, skip to res | x<=y  //op为<=  //op1存x  //op2存y  //res存真假 | 13 | <= |
| If op1>op2, res=1 | x>y  //op为>  //op1存x  //op2存y  //res存真假 | 14 | > |
| If op1>=op2,skip to res | x>=y  //op为>=  //op1存x  //op2存y  //res存真假 | 15 | >= |
| If op1!=op2,skip to res | x!=y  //op为!=  //op1存x  //op2存y  //res存真假 | 16 | != |
| Label op1 //用于表示op1 | label op1： | 17 | label |
| If op1==op2, skip to res | x==y  //op为==  //op1存x  //op2存y  //res存真假 | 18 | == |
| scanf(op1) //op1是int，char均可 | scanf op1 | 19 | scanf |
| printf(op1) //op1是string,int,char均可 | printf op1 | 20 | printf |
| BNZ label //满足条件跳转到label | BNZ label  //label存在op1 | 21 | BNZ |
| BZ label //不满足条件跳转到label | BZ label  //label存在op1 | 22 | BZ |
| Exit //程序结束 | Exit | 23 | exitop |
| Ass | Res = op1 | 24 | ass |

### 7. 目标代码生成方案

数据结构：

Mid\_out🡪FELTI

从中间代码映射到目标代码：

|  |  |  |
| --- | --- | --- |
| 中间代码 | opcon | 目标指令 |
| res = op1 + op2 | + | add res op1 op2 |
| res = op1 – op2 | - | sub res op1 op2 |
| res = op1 \* op2 | \* | mult op1 op2  mflo res |
| res = op1 / op2 | / | div op1 op2  mflo res |
| res[op2] = op1 | []= | Sw op1,op2(res) |
| res = op1[op2] | =[] | Lw res,op2(op1) |
| push x  push y  call fuc  // op1是x,y  //op1是fuc | push | 利用lh，add |
| Call  //无返回值函数调用语句 | 利用lh，add |
| call op1  res = RET | calr  //有返回值函数调用语句 | J op1 |
| ret x  //op为ret  //op1为x | ret | J op1,运行结果存到res寄存器 |
| GOTO label  //op为GOTO,op1存label | GOTO | J res |
| x<y  BZ label  //op为<  //op1存x  //op2存y | <  BZ | t1 = x - y  bgez t1 label |
| x<=y  BZ label  //op为<=  //op1存x  //op2存y | <=  BZ | t1 = x - y  bgtz t1 label |
| x>y  BZ label  //op为>  //op1存x  //op2存y | >  BZ | t1 = x - y  blez t1 label |
| x>=y  BZ label  //op为>=  //op1存x  //op2存y | >=  BZ | t1 = x - y  bltz t1 label |
| x!=y  BZ label  //op为!=  //op1存x  //op2存y | !=  BZ | beq x y label |
| label op1： | label | op1：  表示标签 |
| x==y  //op为==  //op1存x  //op2存y  //res存真假 | == | bne x y label |
| scanf op1 | scanf | 根据类型获取输入 |
| printf op1 | printf | Syscall系统调用输出 |
| BNZ label  //label存在op1 | BNZ | -- |
| BZ label  //label存在op1 | BZ | -- |
| Exit | exitop | 转化为结束程序的系统调用语句 |

### 8. 优化方案

### 9. 出错处理

错误处理分为三类，分别为词法分析错误、语法分析错误和语义分析错误，具体设计如下：

|  |  |  |
| --- | --- | --- |
| 1 | 词法分析 | 解释 |
| 1 | ERROR\_CHAR | 字符不符合文法 |
| 2 | ERROR\_STR | 字符串不符合文法 |
| 3 | ERROR\_INVA | 无效字符 |
| 4 | ERROR\_！ | ！单独出现 |
| 2 | 语法分析 | 解释 |
| 1 | ERROR\_SEMI | 缺少; |
| 2 | ERROR\_WENFA | 与文法规则不匹配 |
| 3 | ERROR\_LPARENT | (不匹配 |
| 4 | ERROR\_LBRACK | [不匹配 |
| 5 | ERROR\_LBRACE | {不匹配 |
| 6 | ERROR\_PARA\_TABLE | 参数表错误 |
| 7 | ERROR\_ARRAY | 数组定义有错 |
| 8 | ERROR\_CONS | 常量定义错误 |
| 9 | ERROR\_IDENT | 标识符错误 |
| 3 | 语义分析 | 解释 |
| 1 | ERROR\_DOWNFL | 数组越下界 |
| 2 | ERROR\_UPFL | 数组越上界 |
| 3 | ERROR\_TYPE | 标识符类型匹配错误 |
| 4 | ERROR\_RE | 标识符重复定义 |
| 5 | ERROR\_NO | 标识符未定义 |
| 6 | ERROR\_MIX\_PARA | 参数不匹配 |
| 7 | ERROR\_ARR\_INDEX | 数组下标类型不匹配 |
| 8 | ERROR\_CONS\_ASS | 给常量赋值 |
| 9 | ERROR\_MIS\_RETURN | 返回值类型错误 |
| 10 | ERROR\_NO\_RETURN | 缺少返回值 |

## 三．操作说明

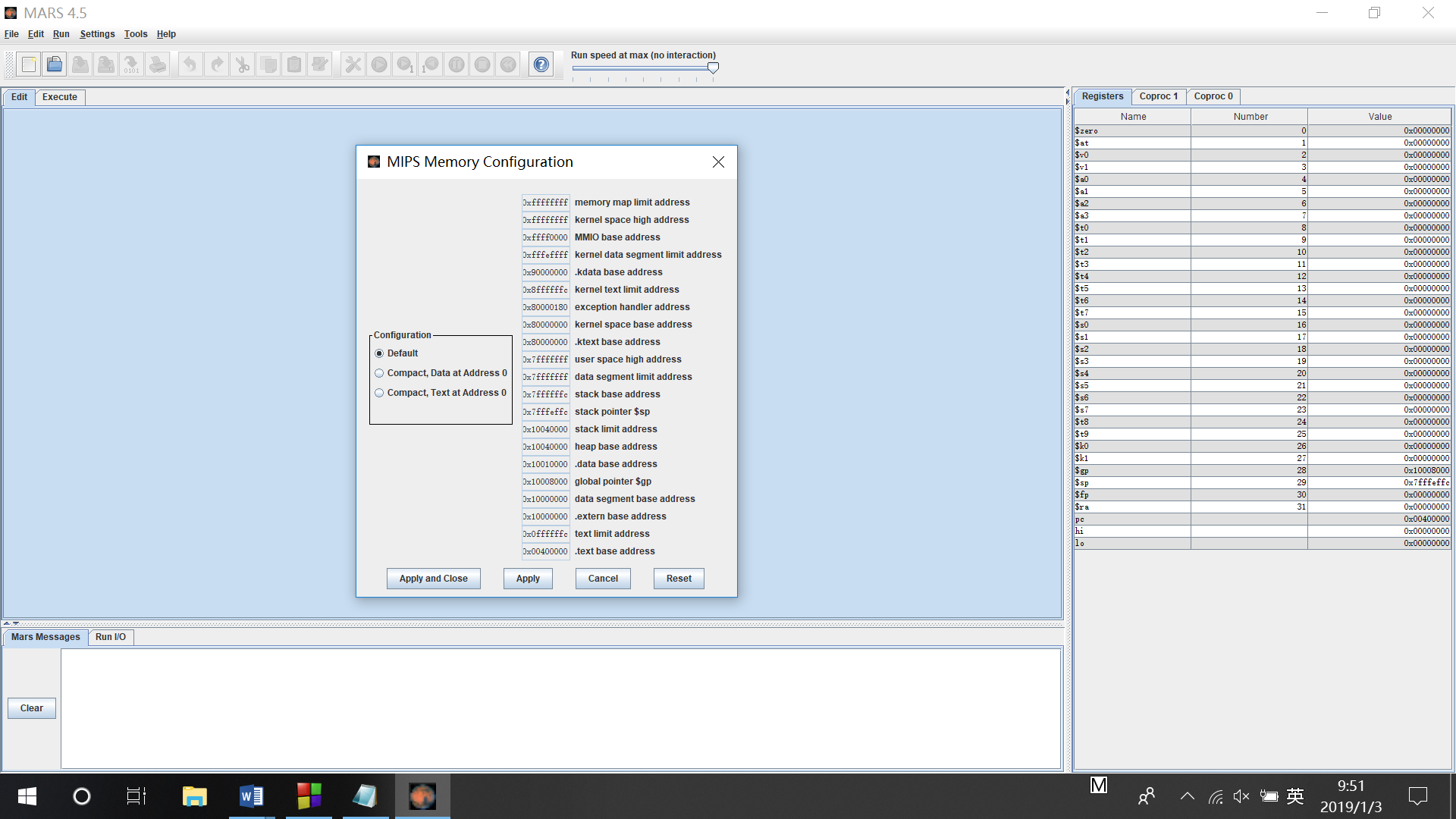
### 1．运行环境

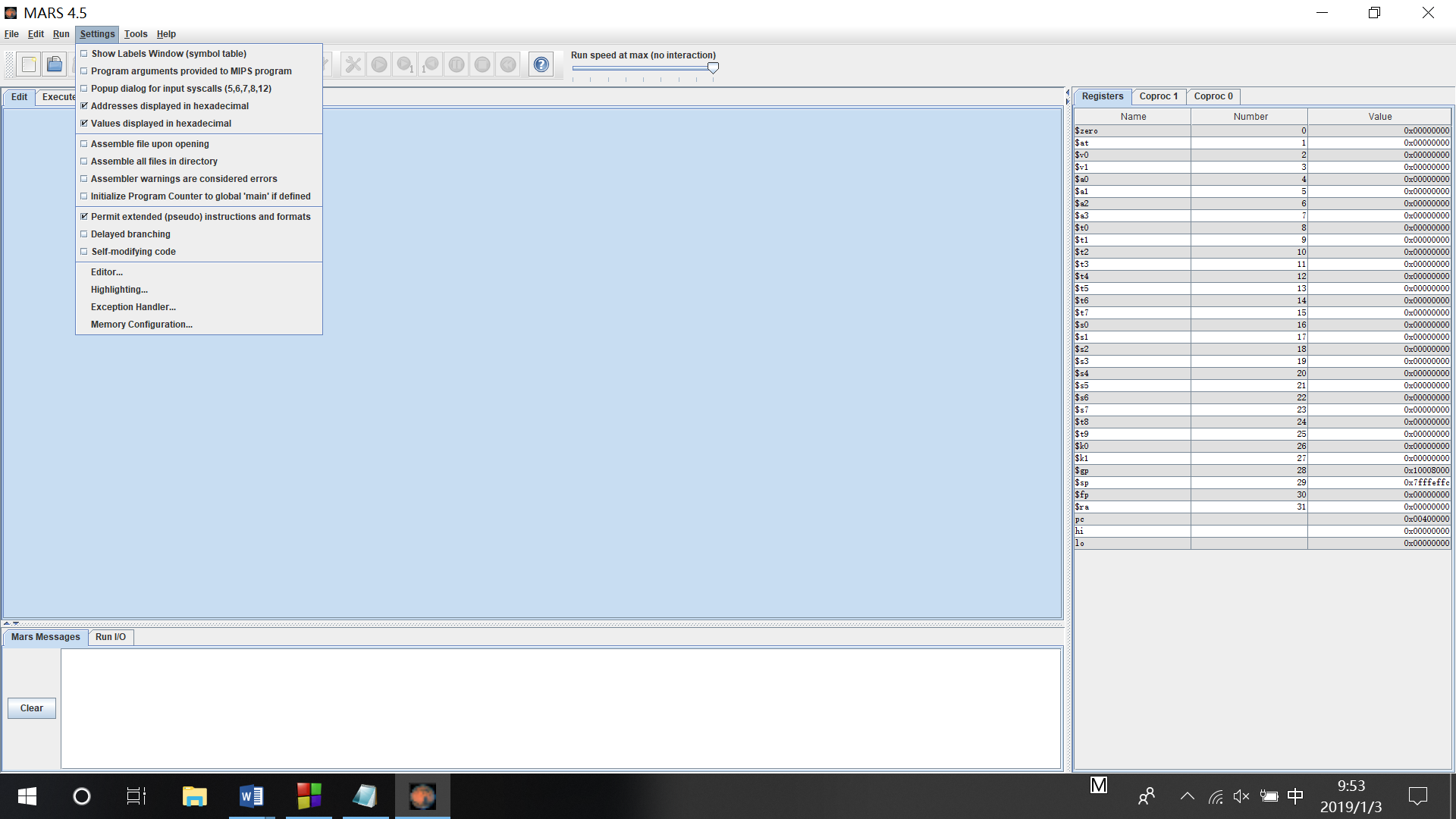
编译程序运行环境：

CodeBlocks16.01

cbp工程文件最好用CodeBlocks打开，main.c文件可以用dev-c++打开。

MARS运行环境：





### 2．操作步骤

运行工程文件后，需要在控制台输入测试文件的路径，在输入完毕后，会在工程文件所在目录下生成7个文件，分别是：

|  |  |
| --- | --- |
| cifa.txt | 词法分析结果 |
| gram\_res.txt | 语法分析结果 |
| \_tables.txt | 符号表输出 |
| \_mid\_code | 优化前中间代码 |
| \_aim\_code.asm | 优化前目标代码 |
| \_opti\_mid\_code | 优化后中间代码 |
| \_opti\_aim\_code.asm | 优化后目标代码 |

## 四．测试报告

### 1．测试程序及测试结果

测试程序见测试程序目录中的文件，下面给出测试程序的运行结果：

1. T1

<-----mainHHhhHH!HHhhHH!<---test2 <AC <=AC >AC >=AC ==AC !=ACtest2---><---test\_ret<--1ret-->2<--2ret-->4test\_ret--->1

2

ab<--1main-->296<--2main-->96<--3main-->98main----->

1. T2

<-----main<---test3<--if\_0\_AC--><--outest\_cons\_char=43--><--if\_char2ascii\_AC-->test3---><--test4\_int-->123<--test4\_char-->c<---test5<--test5\_AC-->test5---><--4main--><---test5<--test5\_AC-->test5--->-26<---test6 !#$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~\n<--test6-->-1234510-123210<--test6-->+-\*/\_morz0379test6--->main----->

1. T3

<-----main<---test7\_1--->1<---test7\_2--->2<---test\_for<--1AC--><--2AC--><--2AC-->test\_for---><---test\_while<--3AC3AC3AC--><--var\_int\_while=3--><--3AC3AC3AC-->test\_while---><---test\_if1\_AC2\_ACtest\_if---><---test\_if\_else<--if\_else\_AC-->test\_if\_else--->main----->

1. T4

<-----main<--5main-->98<--6main-->2960<--7main-->-55<--8main-->207<---test\_add<--1add-->90<--2add-->46test\_add---><---test\_num<--1num-->a<--2num-->-223<--3num-->-11<--4num-->354<--5num-->2test\_num---><---test\_sub<--1sub-->-133<--2sub-->-90test\_sub---><---test\_mult<--0mul-->1<--1mul-->15180<--2mul-->23test\_mult---><---test\_div<--1div-->0<--2div-->2test\_div--->main----->

1. T5

<-----main<---test\_array<--1\_local\_array-->2<--2\_local\_array-->4<--3\_local\_array-->1<--4\_local\_array-->2<--5\_local\_array-->2<--i=4<--out\_a1-->12<--out\_a2-->4--><--i=3<--out\_a1-->6<--out\_a2-->3--><--i=2<--out\_a1-->2<--out\_a2-->2--><--i=1<--out\_a1-->0<--out\_a2-->1--><--i=0<--out\_a1-->0<--out\_a2-->0--><--array\_out\_gcd\_2=2array\_out\_gcd\_2--><--array\_local\_gcd\_1=1array\_local\_gcd\_1-->test\_array---><---test\_char<--test\_char-->97<--test\_char-->98<--test\_char-->-1test\_char---><---test\_gcd12

23

32

32

43

45

65

42

45

7

res[0]=1 res[1]=32 res[2]=1 res[3]=1 res[4]=1 test\_gcd---><---test\_char\_exsa=a'a'=97b=b'b'=98c=99'c'=99d=100'd'=100e=101'e'=101f102test\_char\_exs---><--9main-->!start!<---1calr--->12<---2calr--->26!end!-14<--10main-->!start!<---1calr--->12<---2calr--->26!end!!start!<---1calr--->-111<---2calr--->0!end!-111<--11main--><--1test\_fu\_fuc-->2<--2test\_fu\_fuc-->-2<--3test\_fu\_fuc-->2<--4test\_fu\_fuc-->-2<--5test\_fu\_fuc-->2<--6test\_fu\_fuc-->-21<--12main--><--1test\_fu\_fuc-->2<--2test\_fu\_fuc-->-2<--3test\_fu\_fuc-->2<--4test\_fu\_fuc-->-2<--5test\_fu\_fuc-->2<--6test\_fu\_fuc-->-2-1<--13main-->900main----->

1. F1

line17\_10 :cifa\_error\_ERROR\_CHAR\_DEFINE--.

line18\_12 :cifa\_error\_ERROR\_STRING--?

line18\_13 :cifa\_error\_ERROR\_IVALID\_CHARARCTER--?

line18\_14 :cifa\_error\_ERROR\_IVALID\_CHARARCTER--?

line18\_19 :cifa\_error\_ERROR\_!\_SINGLE--"

line18\_22 :cifa\_error\_ERROR\_STRING--

line18\_4 :grammer\_error\_ERROR\_MIS\_WENFA--

line18\_4 :grammer\_error\_ERROR\_{}\_MIS--

line18\_4 :grammer\_error\_ERROR\_MIS\_WENFA--

line18\_5 :grammer\_error\_ERROR\_MIS\_WENFA--

line18\_6 :grammer\_error\_ERROR\_MIS\_WENFA--hhh

line18\_7 :grammer\_error\_ERROR\_MIS\_WENFA--!

line18\_8 :grammer\_error\_ERROR\_MIS\_WENFA--");"

line19\_1 :grammer\_error\_ERROR\_MIS\_WENFA--}

1. F2

line49\_1 :grammer\_error\_ERROR\_;\_LACK--}

line52\_8 :grammer\_error\_ERROR\_[]\_MIS--)

line60\_1 :grammer\_error\_ERROR\_MIS\_WENFA--void

line60\_1 :grammer\_error\_ERROR\_{}\_MIS--void

line61\_1 :grammer\_error\_ERROR\_()\_MIS--{

1. F3

line4\_4 :grammer\_error\_ERROR\_CONS--;

line21\_10 :grammer\_error\_ERROR\_PARA\_TABLE--)

line30\_4 :grammer\_error\_ERROR\_ARRAY--0

line37\_5 :meaning\_error\_ERROR\_LOW\_ARR--]

line47\_4 :meaning\_error\_ERROR\_HIGH\_ARR--]

1. F4

line47\_19 :meaning\_error\_ERROR\_MIS\_PARA--)

line62\_3 :meaning\_error\_ERROR\_RE\_DEF--main\_cons\_char

line65\_4 :meaning\_error\_ERROR\_MIS\_TYP--;

line66\_4 :meaning\_error\_ERROR\_MIS\_TYP--;

line67\_1 :meaning\_error\_ERROR\_NO\_DEF--aaa

line67\_4 :meaning\_error\_ERROR\_MIS\_TYP--;

* 1. . F5

line25\_10 :meaning\_error\_ERROR\_MIS\_RET--)

line68\_4 :meaning\_error\_ERROR\_CONS\_ASS--;

meaning\_error\_ERROR\_NO\_RETURN--just\_test

### 2．测试结果分析

正确程序主要覆盖情况：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 测试类型 | T1 | T2 | T3 | T4 | T5 |
| 标识符不区分大小写 | √ |  |  |  |  |
| 测试逻辑运算符 | √ |  |  | √ |  |
| 测试函数调用 | √ | √ | √ | √ | √ |
| 测试输入数字 | √ |  |  |  |  |
| 测试输入字符 | √ |  |  |  |  |
| 测试复杂表达式运算 | √ |  |  |  | √ |
| 测试输出函数返回值及类型 |  | √ |  |  |  |
| 测试数组赋值取值 |  | √ |  |  |  |
| 测试字符串 |  | √ |  |  |  |
| 测试输出数字 | √ | √ |  |  |  |
| 测试输出字符 | √ | √ |  |  |  |
| 测试不同函数相同局部变量名 |  |  | √ |  |  |
| 测试for |  |  | √ |  |  |
| 测试while |  |  | √ |  |  |
| 测试if\_else | √ |  | √ |  |  |
| 测试加法 |  |  |  | √ |  |
| 测试数字 |  |  |  | √ |  |
| 测试减法 |  |  |  | √ |  |
| 测试乘法 |  |  |  | √ |  |
| 测试除法 |  |  |  | √ |  |
| 测试局部数组 |  |  |  |  | √ |
| 测试全局数组 |  |  |  |  | √ |
| 测试char型 |  |  |  |  | √ |
| 测试递归调用 | √ | √ |  |  | √ |
| 测试有返回值函数调用 | √ |  |  |  | √ |
| 测试嵌套调用 | √ |  |  |  | √ |
| 测试<表达式>里最前面的+/- |  |  |  |  | √ |
| 测试空语句等 | √ |  |  |  |  |

错误程序覆盖情况

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 错误类型 | 具体错误 | F1 | F2 | F3 | F4 | F5 |
| 词法分析 | ERROR\_CHAR | √ |  |  |  |  |
| ERROR\_STR | √ |  |  |  |  |
| ERROR\_INVA | √ |  |  |  |  |
| ERROR\_！ | √ |  |  |  |  |
| 语法分析 | ERROR\_SEMI |  | √ |  |  |  |
| ERROR\_WENFA |  | √ |  |  |  |
| ERROR\_PARENT |  | √ |  |  |  |
| ERROR\_BRACK |  | √ |  |  |  |
| ERROR\_BRACE |  | √ |  |  |  |
| ERROR\_PARA\_TABLE |  |  | √ |  |  |
| ERROR\_ARRAY\_DEF |  |  | √ |  |  |
| ERROR\_CONS |  |  | √ |  |  |
| 语义分析 | ERROR\_DOWNFL |  |  | √ |  |  |
| ERROR\_UPFL |  |  | √ |  |  |
| ERROR\_TYPE |  |  |  | √ |  |
| ERROR\_RE\_DEF |  |  |  | √ |  |
| ERROR\_NO\_DEF |  |  |  | √ |  |
| ERROR\_MIX\_PARA |  |  |  | √ |  |
| ERROR\_CONS\_ASS |  |  |  |  | √ |
| ERROR\_MIS\_RETURN |  |  |  |  | √ |
| ERROR\_NO\_RETURN |  |  |  |  | √ |

## 五．总结感想

1. 需要合理安排时间。
2. 不要畏难，坚持做下去。
3. 别人的经验不一定适合你，正如别人的方法不一定适合你。
4. 写代码需要一鼓作气，隔三岔五写一些的方法不可取。