# 申优文章

此下为我在C0编译器设计过程中遇到的主要难点：

### 1、变量定义与有返回值函数定义的FIRST集。

**＜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＞

＜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＞‘]’ )}

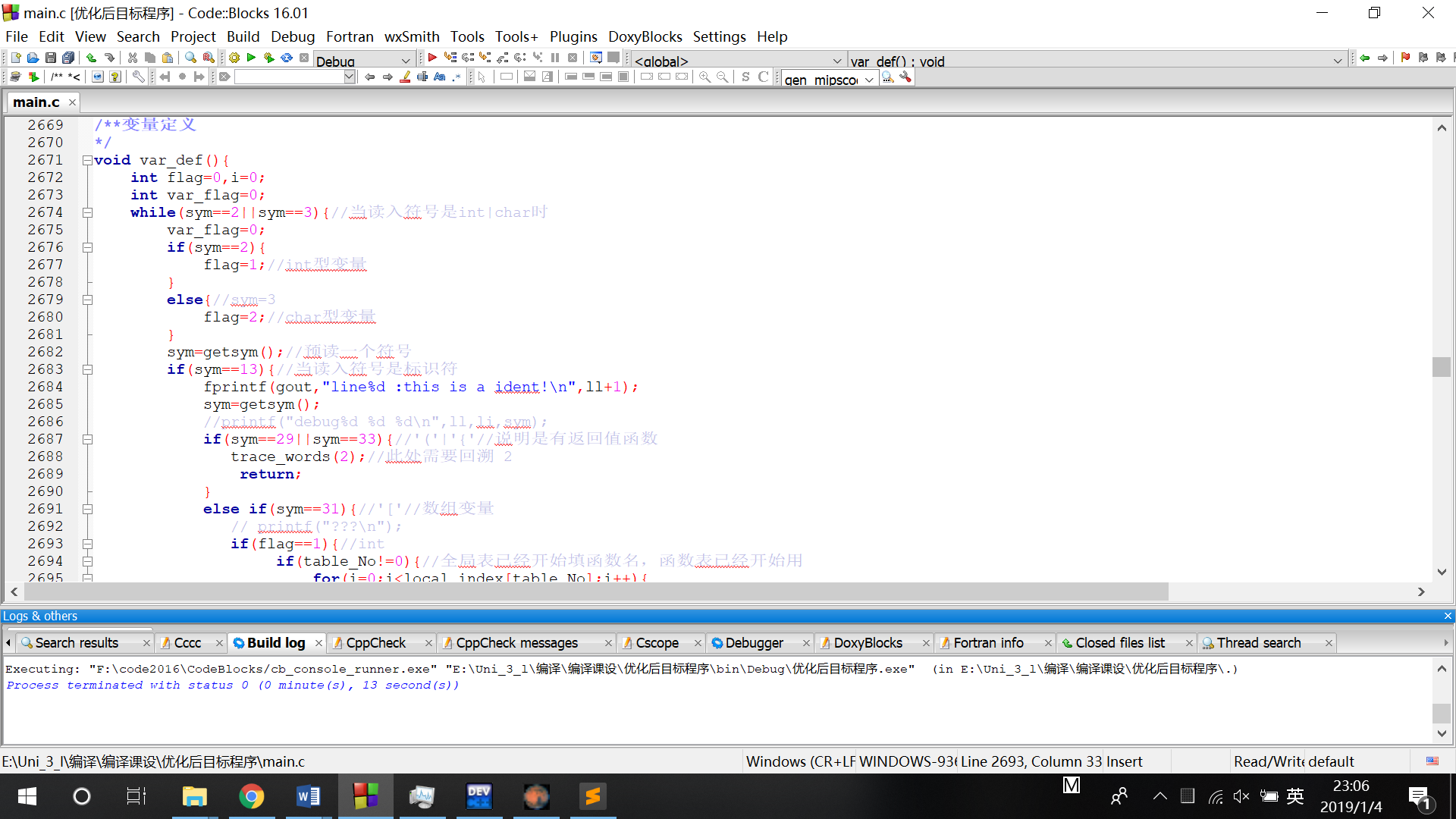
＜类型标识符＞      ::=  **int | char**

＜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＞‘}’

＜声明头部＞   ::= **int＜标识符＞** |**char＜标识符＞**

由文法可以看出<变量说明>的头部与<有返回值函数定义>的FIRST集有交集，此时给识别带来难题。

此处若是一直判断到可以判断该规则是哪条文法规则需要读入至少3位，于是我采用顺序处理方法，默认它是<变量说明>，因为变量说明一定在函数定义的前面。若是不符合<变量说明>，则采用trace\_words()函数回溯2个words，并进入有返回值函数定义。



### 2、<语句>的处理。

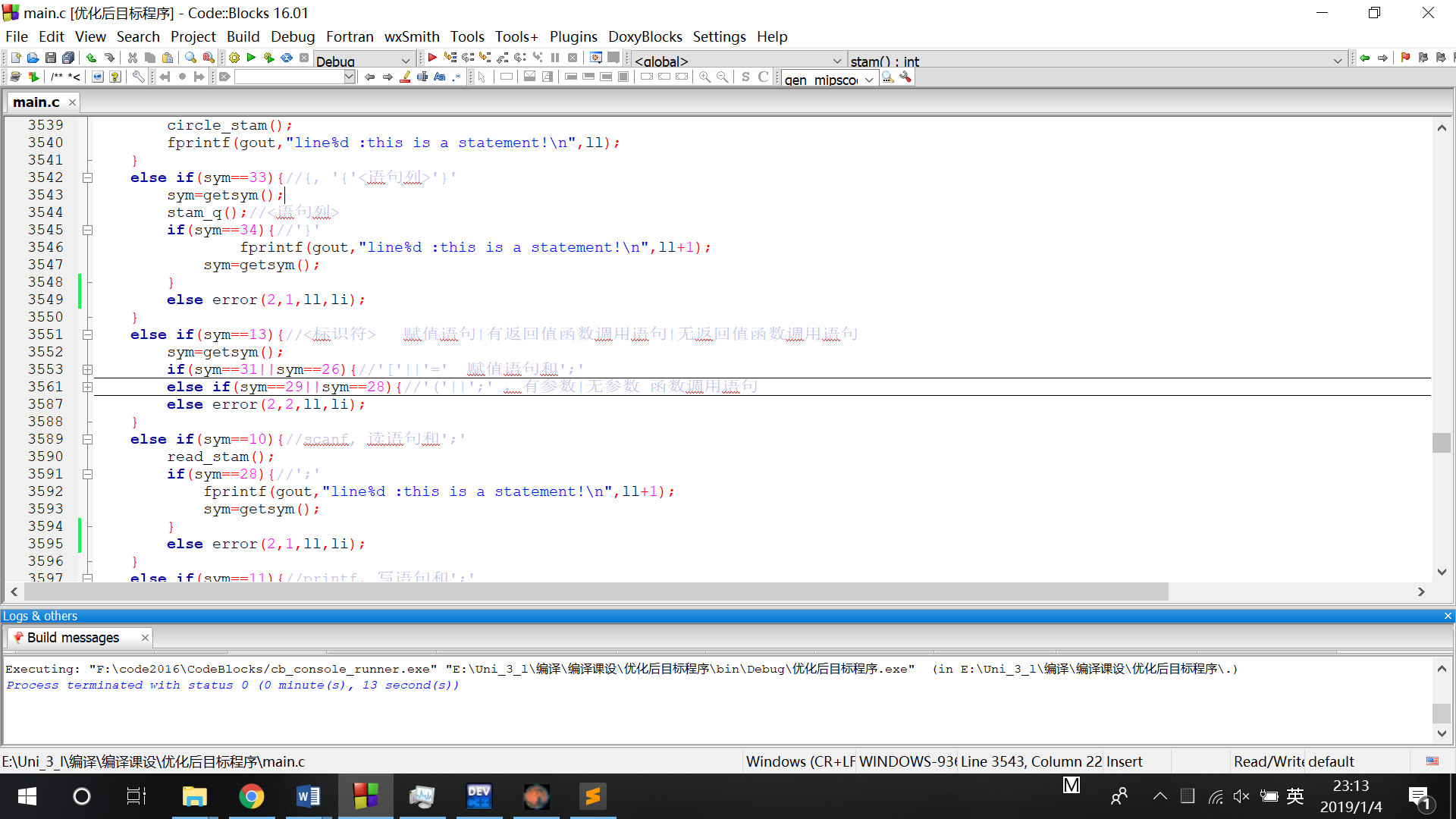
而在<语句>的处理中，情况要更加复杂一些，因为<语句>的规则右部FIRST集相交的情况更多。

不仅仅如此，<有返回值函数调用语句>和<无返回值函数调用语句>的文法规则还是一样的，在语法分析层面无法区分，只能通过语义分析查表得知该语句是有返回值函数调用语句还是无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＞

**＜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



### 3、生成中间代码过程中if，for，while语句的处理。

**if语句**

**源代码：**

if(条件){A}

else{B}

**中间代码：**

条件

BZ label\_else

A

GOTO label\_out

label\_else:

B

label\_out:

**while语句**

**源代码：**

while(<条件>) <语句>

**中间代码：**

label\_cir:

条件

BZ label\_out

<语句>

GOTO label\_cir

label\_out:

**for语句：**

**源代码：**

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

**中间代码：**

<ident>=<exs>

label\_cir:

<stam>

<ident>=<ident>+|-<步长>

<condi>

BZ label\_out

GOTO label\_cir

label\_out:

<ident>=<exs>

记下此时的midcode\_index指针

<condi>-->存到save\_condition里面

<ident>=<ident>+|-<步长>--->存到save\_step里--->一条

返回到上次记下的midcode\_index指针

产生并填入label\_cir

<stam>

取出save\_step

取出save\_condition

BZ label\_out

GOTO label\_cir

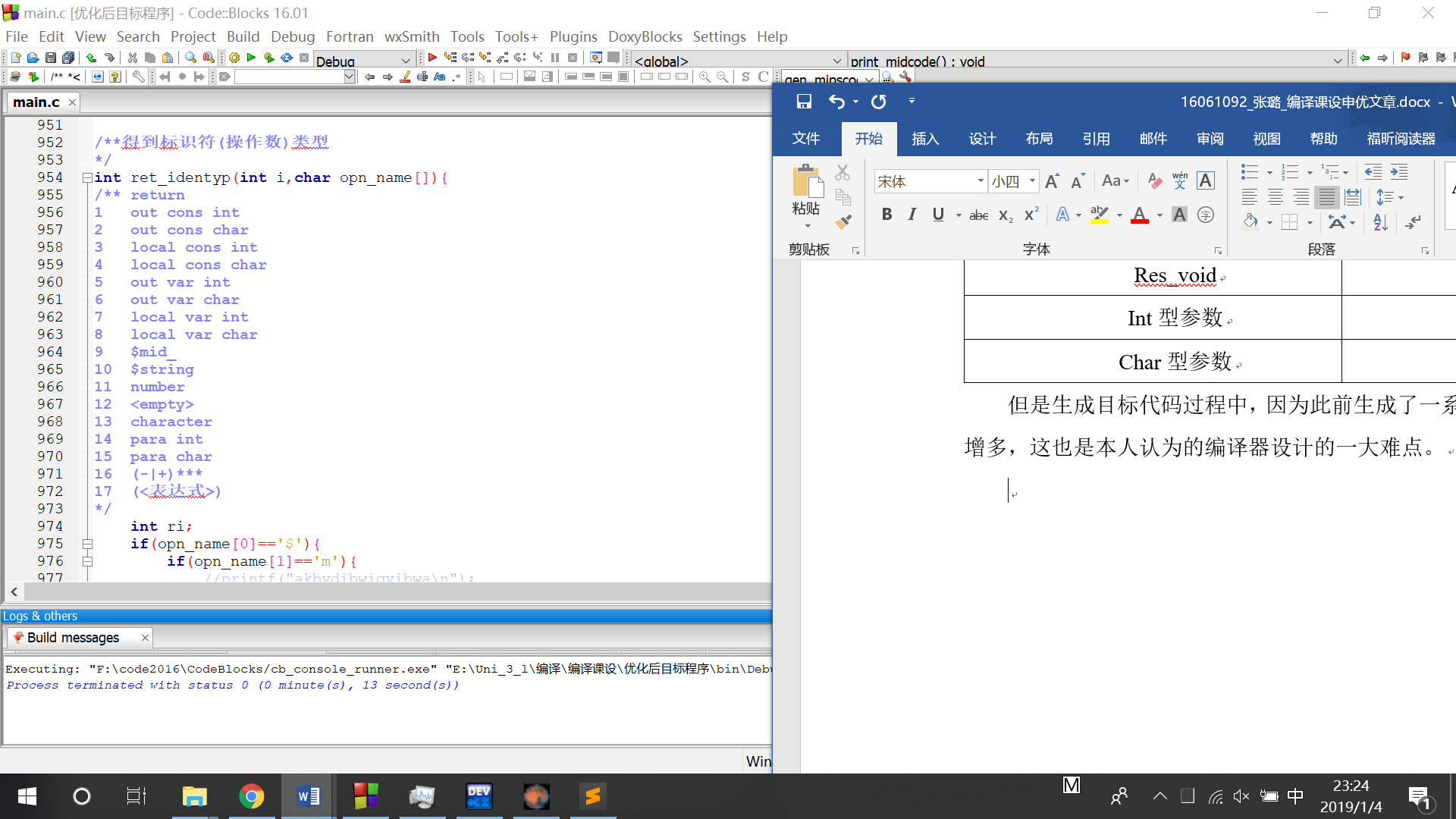
产生并填入label\_out

### 4、生成目标代码时标识符的处理。

源代码的标识符种类有11种，如下：

|  |  |
| --- | --- |
| 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 |
| Res\_void | 9 |
| Int型参数 | 10 |
| Char型参数 | 11 |

但是生成目标代码过程中，因为此前生成了一系列中间变量，标识符的类型增多，这也是我认为的编译器设计的一大难点。



将生成目标代码过程中遇到的标识符类型分为以上17种，并在生成目标代码的过程中分别进行处理。

如下一个将op1 load到$t0的函数：

