**《C0编译器》实验报告**

1. **词法分析**

词法分析是将文件输入的字符流转换成单词流，作为语法分析的输入。单词类型如下：



具体分析见《词法分析实验报告》（第一次实验报告）；

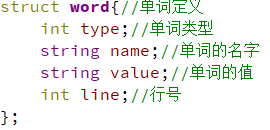
1. **语法分析**

语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等等.语法分析程序判断源程序在结构上是否正确。

本次实验采用的是递归下降子程序法，即为文法中的每一个非终结符定义一个函数，用于分析单词流进入该非终结符的推导式后的结构问题。

**输入：词法分析得到的单词数组**





**读入当前分析的单词：函数含义及其参数含义如图**



各非终结符的递归子程序：

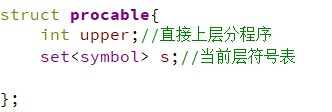


注意：进入各子程序时，待分析单词已经读入，当进入其他子程序或该程序结束时，需要考虑是否需要为下一个子程序读入当前单词。

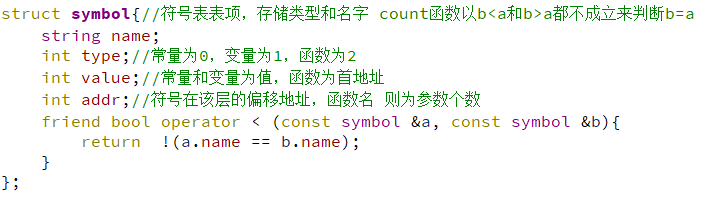
1. **符号表**

符号表是程序在编译阶段生成的结构体，目的是为了记录源程序中的各种名字的特性信息，包括程序名、函数名、过程名以及它们的种类、类型和维数等特性。

**符号表结构**



该结构体为**层级表**，每一层都有一个直接上层，每一层都不能含有和直接上层（或直接上层的直接上层，直到最外层）的符号同名的变量定义；



**需要注意的是，对于变量和常量的符号，value字段存储它的数值，addr存储它在该层的偏移地址；对函数名的符号，value保存函数的首地址，方便函数调用时的跳转，二addr保存函数的参数个数，方便函数的传参。**

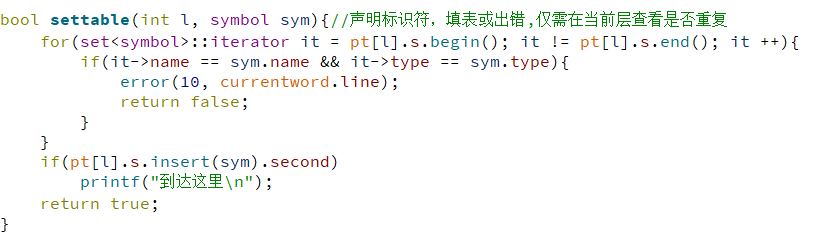
**另外，由于STL set中不能存在含有相同key的键值对，而这一特性是由该容器的运算符“<”判断的：当set中a<b且a>b都为假时，认为a=b，则无法insert。所以，我们先将小于运算符进行重载，以保证符号名相同时不能插入。**

**符号表管理**

同时，我们需要为符号表的管理定义几个函数：



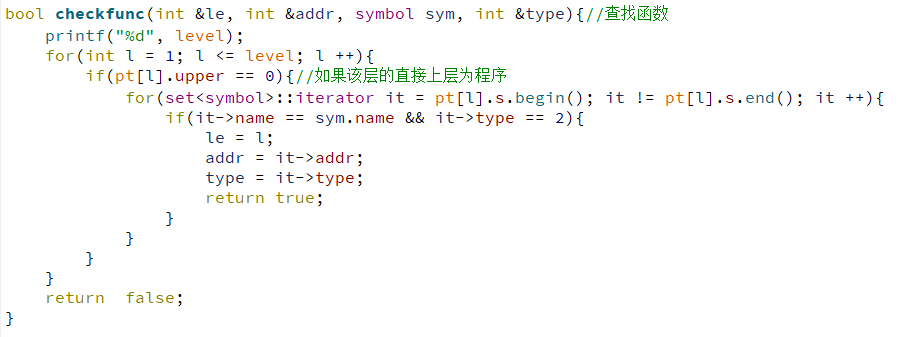
**查表操作**，在当前层l及其上层查询符号sym，并返回查询到的符号的层级le，addr字段addr，以及类型type；



**填表操作**，首先查询当前层是否有和sym同名的符号，如果没有，则将sym填入该层符号表；



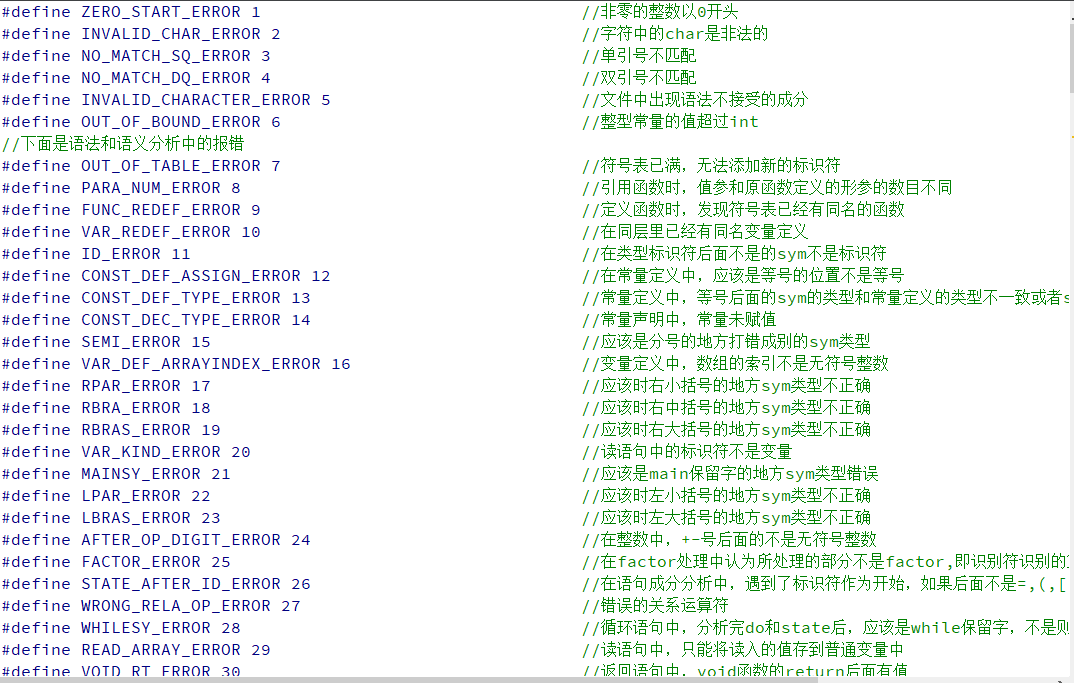
**重填符号表操作**，该操作发生在赋值操作后，将赋值过的sym重填入对应层级的符号表中，需要注意的是，由于STL set容器中存入的是const数据，已经填入的结构体不能直接更改其字段，所以需要先将待重填的符号删除，再将重填符号插入；



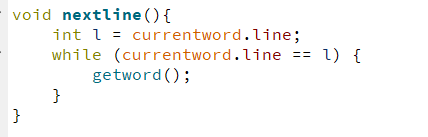
**查找函数名操作**，由于从所有函数名开始，层级号都增加一，且所有函数层的直接上层都为最外层（0层），所以在所有以0层为直接外层的层级中查找函数名的定义，并将函数的层级le及参数个数addr返回；

1. **错误处理**

编译时需要尽可能多的发现程序的编译时错误，良好的编译器总是需要准确的报出错误的类型和行数，一下是部分错误类型定义：

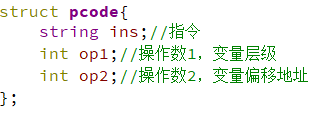


由于一个错误的产生往往会导致产生错误的单词的左右也会有错误产生，所以这里我采用的方法时，如果遇到某个分析单词出现错误时，首先报错，之后**直接换行**，再读取下一个单词：



1. **语义分析与pcode代码生成**

语义即为某个语句需要执行什么操作，生成pcode代码则是通过该种已经定义过的简单中间伪代码来描述语句执行的操作和涉及的操作数（通常为双操作数）：



**需注意：变量定义、常量说明等步骤仅包含查表、填表等，并不生成中间代码！**

**条件语句的地址回填：**

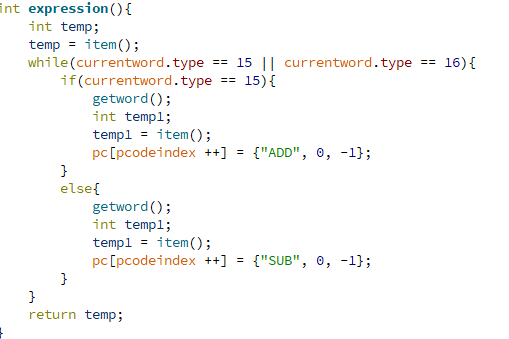


**While语句的地址回填：**



**部分条件表达式的pcode生成：**

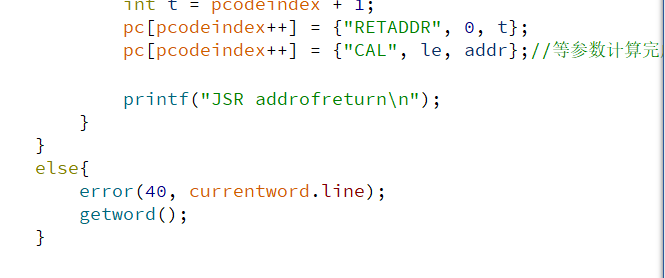




**函数调用的pcode生成：**

函数调用与返回的pcode生成应该是c0编译器的最难的部分。我新添了指令“RETADDR”，作用是通过地址回填的方法，、将函数的返回地址（即CAL指令的下一条）压入栈中，之后再执行CALL指令，是PC转向调用函数的起始地址，即调用函数名的value字段：





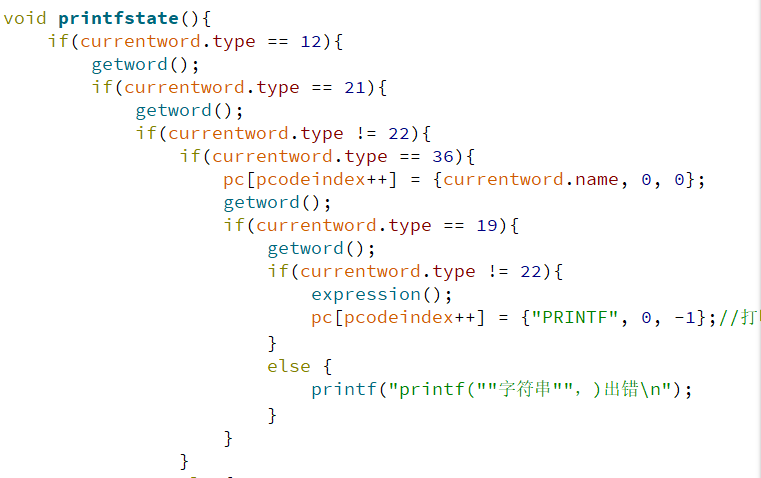
**函数返回：**

函数返回时，则执行RET指令，即使程序计数器跳转向栈顶的值，这需要与RETADDR指令以及CAL指令管理好入栈出栈的顺序（在解释程序中可以看到）：



**打印程序：**

由于C0文法中可以打印字符串，于是我将指令集做出一些修改：如果指令名为PRINTF，则将栈顶的值打印出来，如果指令名不为上述所有指令，则将该指令名打印出来，并通过这种方法打印字符串：

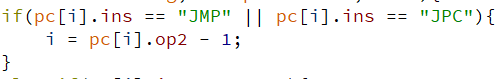




1. **解释程序**

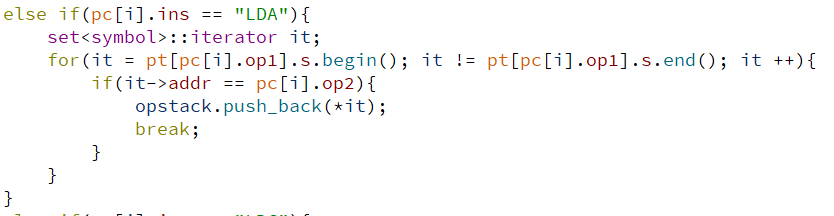
**跳转指令：JMP、JPC**

直接让程序计数器PC指向跳转的地址



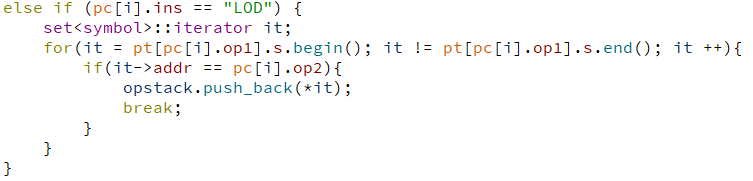
**加载地址：LDA**

将op1层级，偏移地址为op2的变量地址加载到栈顶



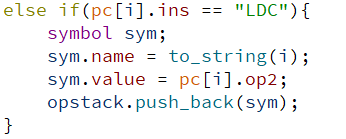
**加载变量：LOD**

将op1层级，偏移地址为op2的变量加载到栈顶



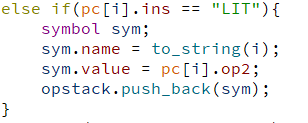
**加载整数：LDC**

将数值为op2的整数加载到栈顶



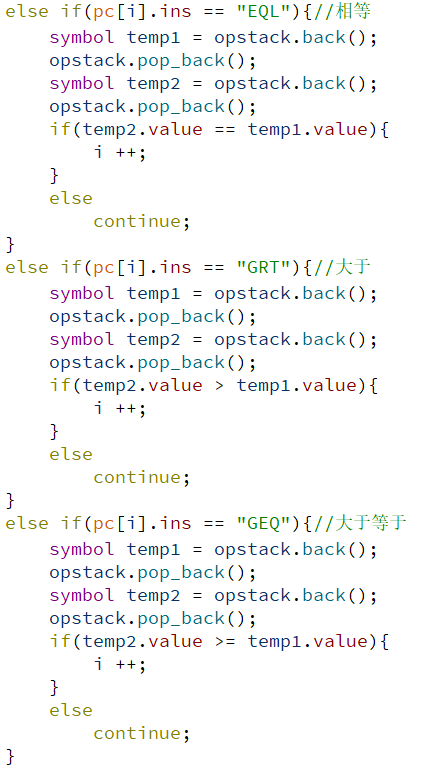
**加载常量：LIT**

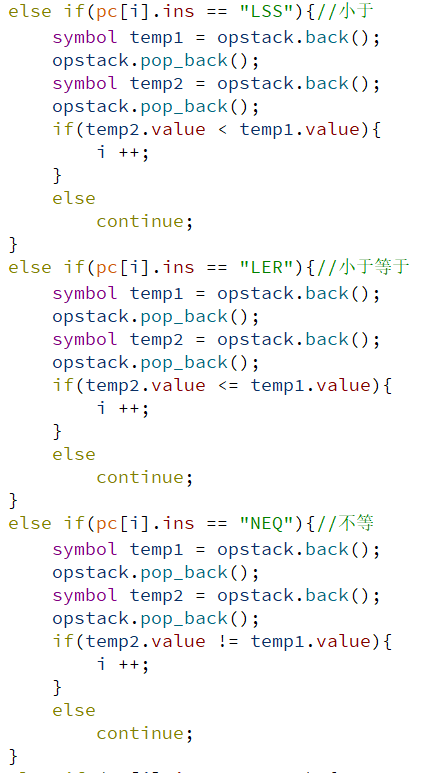
将常量的数值压入栈顶



**条件比较：EQL、GRT、GRQ、LSS、LER、NEQ**

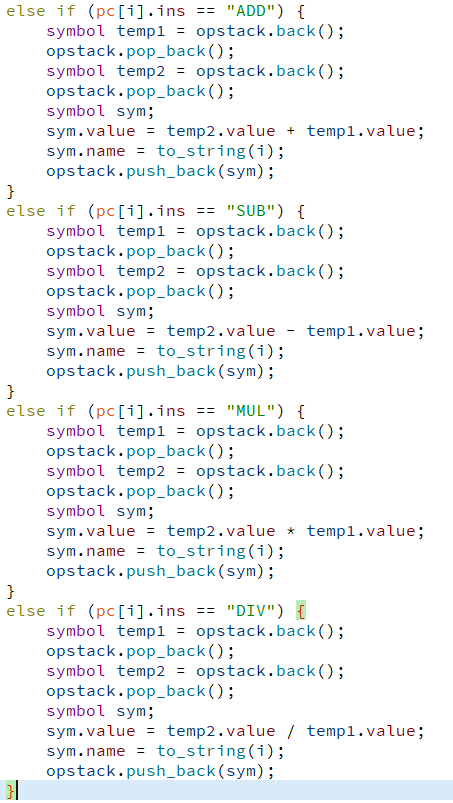
将栈顶元素（右操作数）和次栈顶元素（左操作数）弹出，并做相应比较，并将结果传给下一条JPC指令：





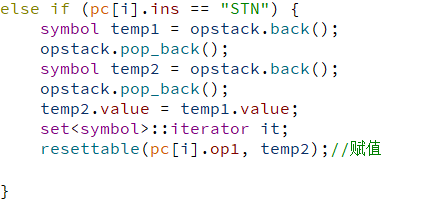
**算术运算：ADD、SUB、MUL、DIV**

将栈顶元素（右操作数）和次栈顶元素（左操作数）弹出栈，执行相应运算后，将结果再压入栈：



**赋值指令：STN**

将栈顶元素存入次栈顶的地址中，并将栈顶和次栈顶弹出栈



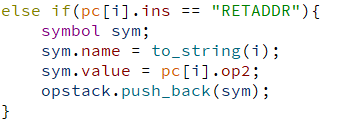
**函数调用：CAL**

此时返回地址存在栈顶，函数的参数逆序存在次栈顶往下，所以需要栈顶的实参值存入函数的形参中（形参地址在声明时已经定义为从0开始），之后再将程序计数器PC指向函数名的value字段：



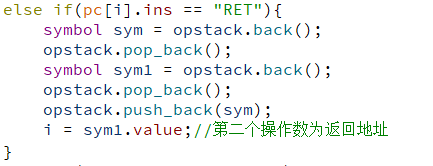
**函数返回地址入栈：RETADDR**

将返回地址op2压入栈中：



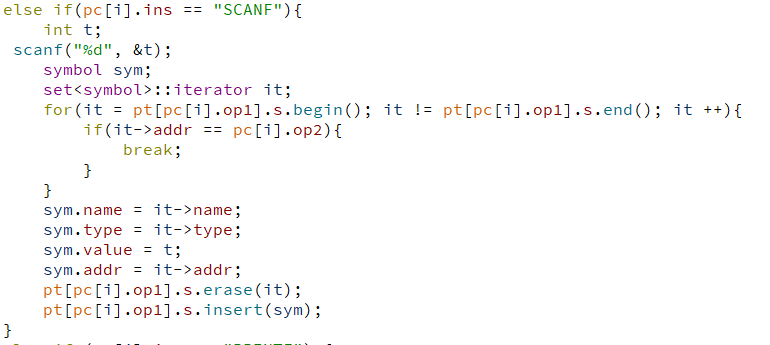
**函数返回：RET**

函数返回时，栈顶为返回值，次栈顶为返回地址，所以将次栈顶出栈，并将程序计数器指向返回地址：



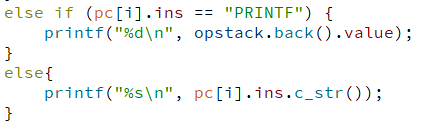
**程序输入：SCANF**

将输入的值存入栈顶地址



**程序输出：PRINTF或其他**

输出栈顶的值或者字符串



1. **实验感想**

**其实编译器的结构还是很清晰的：递归子程序只是简单的递归问题，符号表负责符号的存储和检查工作，错误处理程序负责在判定错误的时候输出错误类型和行号，同时在递归子程序中隐含的插入一些类似属性翻译的操作，来生成对应的pcode和所需要的参数，并在最后将所有pcode收集起来，在符号表的支撑下，解释整个程序的运行过程。写实验报告觉得难度不大，结构也很清晰，只有真正的完整做下来，才能发现这其中又有多少数不尽的坑和乱七八糟的问题。两个星期的抓耳挠腮，平均熬夜到一两点，甚至在图书馆差点晕厥。不过我也非常庆幸，在忙了这么多天后终于有了个完美的结局，过程中也是学到了很多。**