编译原理语义分析及中间代码生成实验报告

选课序号： 19 专业班级： 计算机科学与技术2019-3班

姓名：\_胡聿鑫\_学号：\_ 2220192813\_\_\_\_\_\_日期： 2021年11月28日

# 实验目的

在实现词法、语法分析程序的基础上，编写相应的语义子程序，进行语义处理，加深对语法制导翻译原理的理解，进一步掌握将语法分析所识别的语法范畴变换为中间代码的语义分析方法，并完成相关语义分析器的代码编写。

# 实验内容及要求

采用语法制导翻译等方法实现语义分析程序。生成中间代码或目标机代码。

（1）能发现语义错误，并将错误信息输出到屏幕上，自定义错误处理模式；

（2）生成中间代码或目标机代码。

# 实验相关理论知识

首先给出语义规则的定义：属性计算的过程即是语义处理的过程，对于文法的每一个产生式配备一组属性的计算规则，则称为语义规则。

语义分析负责检查程序的上下文相关的属性，这种属性和具体的语言相关，如：变量在使用前应当声明，表达式均要有合适的类型，函数调用和函数的定义要一致……

在语义分析过程中，需要注意以下几点：

（1）终结符只有综合属性，它由词法分析器提供。

（2）非终结符既可以有综合属性也可以有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值。

（3）产生式右边符号的继承属性和产生式左边符号的综合属性都必须提供一个计算规则。

（4）产生式左边符号的继承属性和产生式右边符号的综合属性由其它产生式的属性规则计算。

语义分析是基于语法分析程序进行的，运用递归下降语法制导翻译法，修改语法分析器，在语法分析部分插入语义动作。递归下降法的原理是利用函数之间的递归调用来模拟语法树自上而下的构建过程。从根节点出发，自顶向下为输入串中寻找一个最左匹配序列，建立一棵语法树。对于每个非终结符的继承属性当作形式参数，函数的返回值当作非终结符的继承属性；对于终结符要初始化所有的继承属性。再进行分析过程中，非终结符根据当前的输入符号决定使用哪个产生式候选。

对每个非终结符A构造一个函数过程，对A的每个继承属性设置一个形式参数，函数的返回值为A的综合属性（作为记录，或指向记录的一个指针，记录中有若干域，每个属性对应一个域）。

非终结符A 对应的函数过程中，根据当前的输入符号决定哪个产生式候选。

每个产生式对应的代码中，按照从左到右的次序，对于单词符号（终结符），非终结符和语义分析分别做以下的工作：

（1）对于带有综合属性x的终结符X，把x的值存入为X.x设置的变量中。然后产生一个匹配X的调用，并继续读入一个输入符号。

（2）对于每个非终结符号B，产生一个右边带有函数调用的赋值语句c=B(b1,b2,…，bk)。

（3）对于语义动作，把动作的代码复制到分析器中，用代表属性的变量来代替对应属性的每一次引用。

为了更加清晰地表达语义分析的结果，可以引入符号表。符号表用来存储程序中的变量相关信息，如类型、作用域、访问控制信息等等。为了使得符号表尽可能高效，可以使用散列表的数据结构来实现，这样查找的时间复杂度仅为O（1），为节约空间，也可以使用红黑树，但时间复杂度会提升为O（lgN）。用符号表处理作用域时可以采用以下两种方法：

（1）采用一张表的方法：进入作用域时插入元素，退出作用域时删除元素。

（2）采用符号表构成的栈：进入作用域时插入新的符号表，退出作用域时删除栈顶符号表。

# 实验过程、运行结果

语义分析程序首先需要经过词法分析和语法分析，基于前两次的实验加以改进。若未通过词法和语法分析，均要显示相应的错误。在实验要求的基础上，额外实现了打印符号表的功能。

实验中能够识别的所有抽象机代码如下：

**LOAD D 将D中的内容加载到操作数栈。读变量**

**LOADI 将常量压入操作数栈**

**STO D 将栈顶单元内容存入D，且栈顶单元出栈。存变量**

**ADD 将次栈顶与栈顶单元出栈并相加，和置于栈顶。**

**SUB 将次栈顶单元减去栈顶单元并出栈，差置于栈顶。**

**MULT 将次栈顶与栈顶单元出栈并相乘，积置于栈顶。**

**DIV 将次栈顶与栈顶单元出栈并相除，商置于栈顶。**

**BR lab 无条件转移到lab**

**BRF lab 若栈顶单元逻辑值，假(0)则转移到lab**

**EQ 将栈顶两单元做相等比较，并将结果真或假(1或0)置于栈顶**

**NOTEQ 栈顶两单元做不等于比较，并将结果 (1或0)置于栈顶**

**GT 次栈顶大于栈顶操作数，则栈顶置1，否则置0**

**LES 次栈顶小于栈顶操作数，则栈顶置1，否则置0**

**GE 次栈顶大于等于栈顶操作数，则栈顶置1，否则置0**

**LE 次栈顶小于等于栈顶操作数，则栈顶置1，否则置0**

**IN 从标准输入设备(键盘)读入一个整型数据，并入操作数栈**

**OUT 将栈顶单元内容出栈，并输出到标准输出设备上(显示器)**

**STOP 停止执行**

**CAL 函数调用**

**ENTER 在栈中为函数调用开辟数据区**

**RETURN 从函数返回**

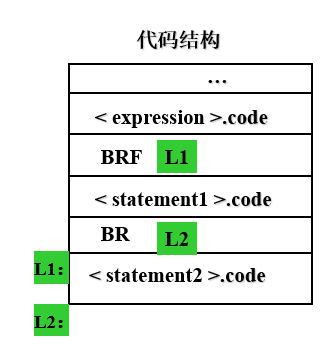
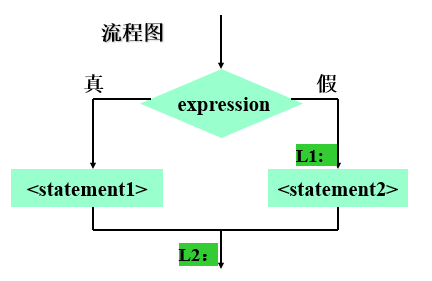
**符号表的实现**：首先定义一个结构体数组，其中的内容包括序号，变量名及其类型，变量地址这四项内容。对于变量的识别，主要有函数名和变量名两种类型。函数名的识别方式为：程序扫描到function 或main时，将function后定义的函数名或main加入到符号表中。对于定义的函数名地址，每当定义一个新的函数，其相对地址都为1，函数内定义的局部变量的相对地址从2开始。Main函数的地址比较特殊，需要当程序执行到main函数入口时返填。由于程序只能识别int这一种数据类型，因此可以进行一定的简化，每当程序扫描到一个int类型的变量时，将序号+1，其名称和地址记录到符号表中。为了区别函数名与变量名，可以简单定义一个布尔函数进行判断，若识别到变量的地址为1或变量名=main，则返回1，否则均返回0，这样一来，所有值为1的变量均为函数，而为0的变量均为普通的局部变量。

整个实验流程图如下：



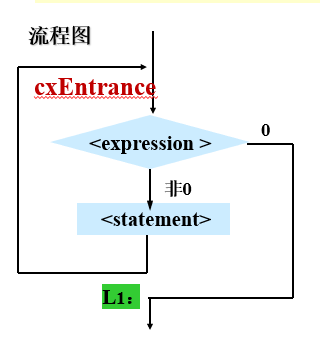
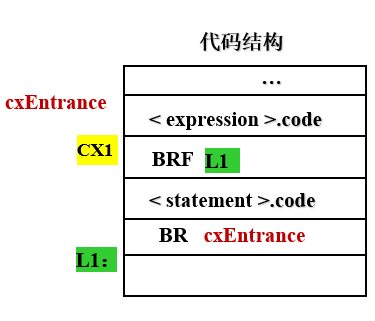
实验中尤其要注意if语句、while语句和for语句在何时进行假转，何时无条件转移，为此首先给出三种语句的流程图及代码结构。

**if语句：**

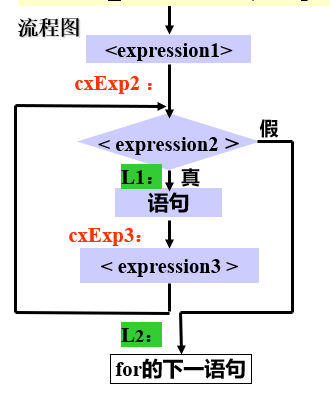
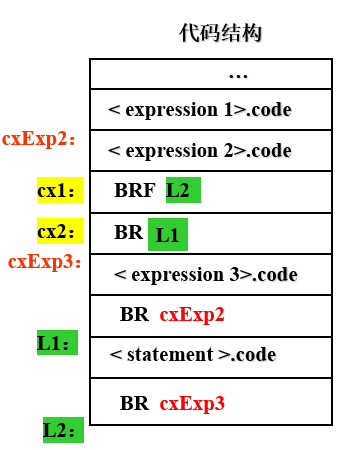


（需要注意的是，若if语句为单分支结构，则L1和L2是相等的）

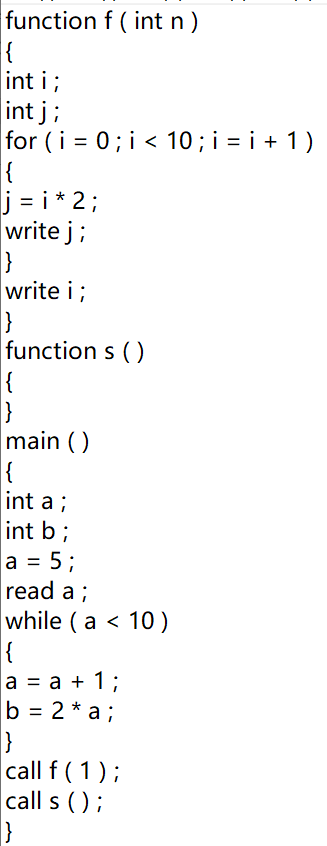
**while语句：**

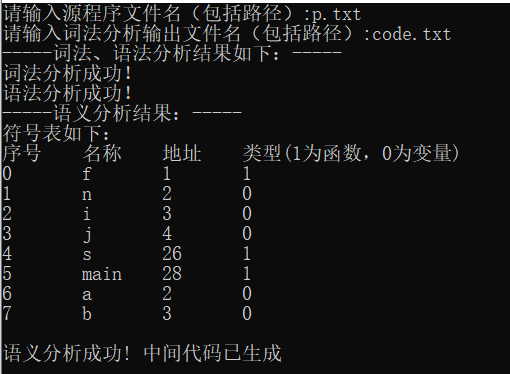
**for语句：**

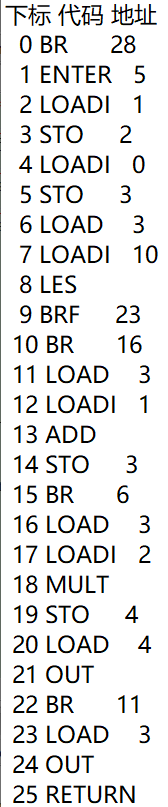
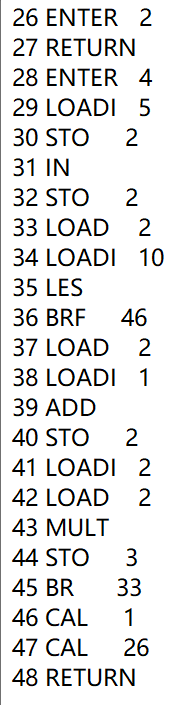
为了尽可能全面体现实验实现的功能，编写了如下TEST源程序：



该程序包括对for和while循环的运用，定义了含参数的函数调用和不含参数的函数调用，也包括了赋值、大小比较等算术逻辑运算操作，对于变量的读写也囊括其中，运行后得到的结果如下：

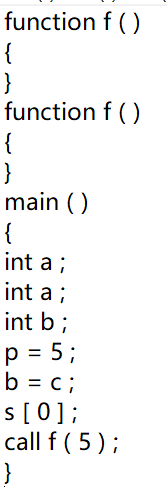


从TEST程序中可以看到，共有main函数加上两个自定义函数，中间代码每进入一个函数时都会执行ENTER，并且结束时执行RETURN。因此中间代码应当有三个ENTER+RETURN，其中间的部分即为函数体相关内容。为方便表示偏移地址，下标的数字即为对应代码的偏移地址。ENTER后的值由函数体内定义的局部变量个数决定，默认值为2，每定义一个局部变量则+1（ENTER决定了函数调用时需要开辟空间的大小）。最终得到的结果如下：

除了实现正确输出给定的中间代码外，该程序还可以实现**连续查错并定位错误位置**，同时统计错误个数。具体实现方式为：扫描程序的过程中，发现至少1个语义错误后，便不再生成中间代码，转而使用预先定义的一个全局变量char数组，作为一个临时的空间，在这个临时空间中，存储该错误（要显示错误类型只需加上一个printf语句并将错误以%s的形式输出即可）。完成存储该错误后，不会立即停止运行，而是继续向下扫描直至结束或再次遇到错误。如此反复，遍历整个程序后，将错误逐个输出。要定位错误所在位置，只需定义一个全局变量并将其初始化为1，每扫描完一行便+1，当输出错误时将此时该变量的计数一并输出，即可得到错误所在行数。最后想统计错误的个数，只需要在扫描完整个程序后，统计上述char数组中非空格的个数，即可得到错误总数（char数组以全局变量的形式定义时，若不进行初始化，则其所有元素均默认为‘ ’，即一个空格）。

为便于调试，现将所有可识别的语义错误均编写在同一个TEST程序中，如下图所示：



本程序共能够识别的6种错误如下：

（1）**重复定义函数名**：TEST程序中两次定义function f()；

（2）**重复定义变量名**：TEST程序中两次定义int a；

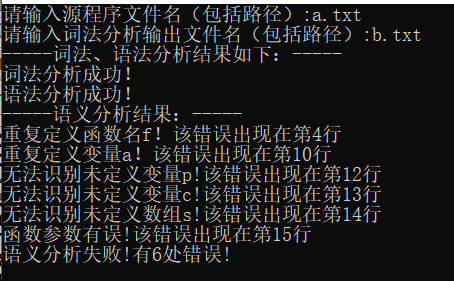
（3）**未定义变量无法使用**：TEST程序中给未定义的变量p赋值；

（4）**无法将未定义变量赋值给其他变量**：TEST程序中将未定义的变量c赋值给已定义的变量b；

（5）**无法使用未定义的数组**：TEST程序中S[0]这一数组没有进行定义；

（6）**错误使用函数调用的参数**：TEST程序中自定义函数f并没有定义参数，而主函数中调用f时非法使用了参数5。

运行后的结果如下图所示：



该结果将对应错误和其所在行数均正确输出，并且给出了错误的总个数，与预期结果一致。

# 实验总结

通过这次实验，我对语义分析的相关基本知识进行了较为全面的回顾，从词法分析到语法分析再到语义分析，我对自己的代码不断进行优化，在保证效率的同时尽可能地降低算法的复杂度。本实验需要在语法分析程序的基础上进行改动，在function\_body，bool\_expression，addi\_expression，factor和term等等所有语法分析中已经成功实现的语句中，均要输出相应的中间代码，具体的实现方法是：首先和语法分析相同，利用strcmp函数对token和对应的字符进行比较，若一致则利用strcpy函数将对应的中间代码输出，例如扫描到“+“号：

if (strcmp(token, "+") == 0)

strcpy(codes[codesIndex++].opt, "ADD");

**连续查错实现心得**：语法分析在每个语句最后均让变量es在大于0时返回，这样便无法实现连续查错，一旦遇到错误便会停止，为了实现连续查错，我对代码进行了改进，判断到es>0时不再return es，而是进行错误判断，输出对应的错误和所在行数。当然，并非对所有语句都需要进行错误判断，而是要根据具体的语义错误而定。例如，在表达式语句中，只可能出现未定义变量或数组这类错误，而不会出现函数名重复定义或未定义的错误，也不可能出现其他重复定义类的错误，因为在一个表达式语句中只可能有一个变量定义语句，重复定义类的错误至少需要两条语句。因此，错误的处理并不需要在所有语句的函数定义中实现，只需要在特定的语句中进行。在<表达式>，<因子>中主要出现未定义变量和未定义数组这两种错误，read语句中在采用键盘输入时也可能出现无法识别未定义变量或读入的数这两种错误，而call语句可能出现的错误主要为未定义函数名或函数参数错误，至于重复定义变量、函数名、数组这类错误，则主要出现在插入符号表的过程中。

在语法分析程序的设计过程中，我没有实现连续查错和错误定位的功能，因此我便始终将其当成我的努力目标。在课后，我学习了网上大量关于错误处理方面的代码，并最终在语义分析的程序中实现了连续查错，也能够给出错误所在行数及错误总数，弥补了我在语法分析程序中的遗憾。

此次实验也依旧存在不足之处，由于只插入一张符号表，在进入作用域时插入元素，退出作用域时删除元素，因此无法实现对全局变量的定义，只能在函数体内定义局部变量。若能够采用符号表构成的栈，构建多张符号表，则可以实现全局变量的定义，相关的查错功能也能够随之实现。同时，本程序只实现了6种比较常见的语义错误，但还有多种语义错误是程序无法识别的，这也需要日后进一步地完善。

从词法分析到语法分析再到语义分析，我对于编译原理整体的课程设计思想有了更深的理解，我意识到光学好理论知识是远远不够的，实践过程中不仅仅是考验自身对理论知识的掌握，更是对计算机科学的全方位考核，仅仅盲目照抄书后代码确实可以实现最基本的功能，却远不能达到老师设置此课程的真正目的。我认为，不仅要对编译原理相关的理论知识有较为扎实的掌握，还要能够熟练运用多种数据结构和算法，并且自己本身需要逐渐培养出一种计算机的设计思维，这样才能真正地自主编写出完善的程序，实验过程中我也充分认识到了自己的不足，今后我会更加努力，不断拓展自身能力，巩固自身理论知识。