



**本科学生实验（实践）报告**

**院 系： 计算机学院**

**实验课程：编译原理实践项目**

**实验项目：Mini C编译器的实现**

**指导老师：黄煜廉**

**开课时间：2019 – 2020 年度第 2 学期**

**专 业：计算机科学与技术**

**班 级：.**

**姓 名：BEAR**

**学 号：.**

**华南师范大学教务处**

**一、实验内容**

实现Mini C语言编译器和虚拟机，编译步骤包括词法分析、语法分析、语义分析（主要是符号检查和类型检查）和代码生成，编译程序要提供显示、保存语法树和目标代码的功能。代码生成使用TM机指令集，虚拟机程序可以在TM机的基础上进行修改。

**1. Mini C的词法**

（1）关键字：else if int return void while

（2）专用符号：

+ - \* / < <= > >= == != = ; , ( ) [ ] { } /\* \*/

（3）标志符和数字

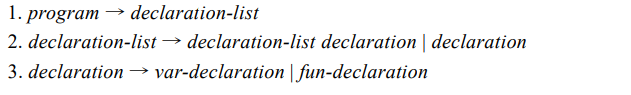
ID = letter letter\*

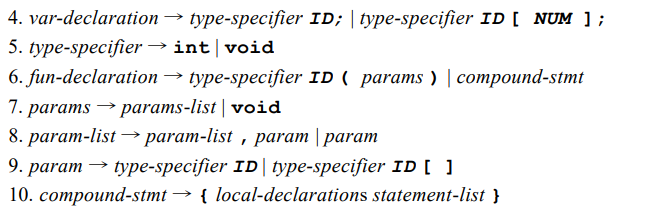
NUM = digit digit\*

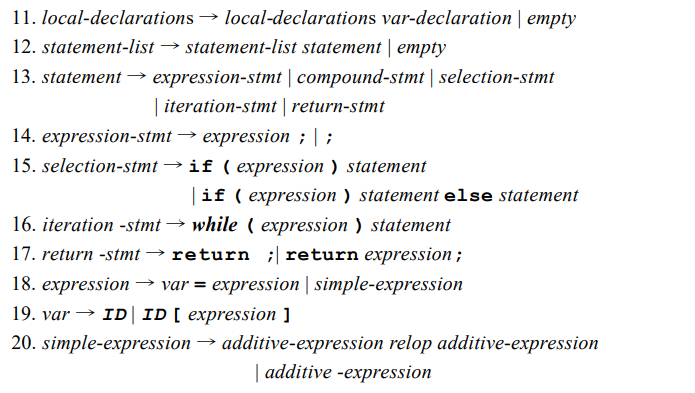
letter = a|..|z|A|..|Z

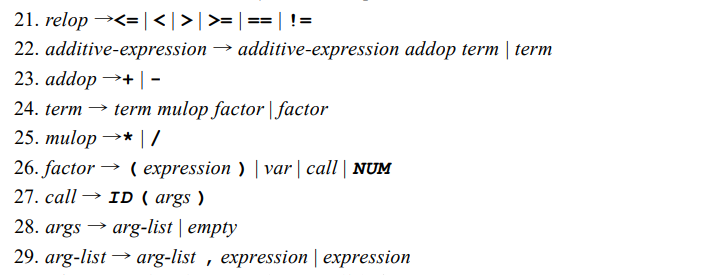
digit = 0|..|9

**2. Mini C的文法规则**









注：第6条语法应改为



**3. Mini C的语义分析**

Mini C的语义主要有如下几点：

（1）标识符在使用前必须已经声明。

（2）在同一个作用域下，同一标识符不能被重复声明。

（3）最后一个声明必须是一个名称为main的函数。

（4）若函数返回值类型为int，则必须存在return语句。

（5）条件表达式的类型必须为int；赋值符号、条件运算符两边的表达式类型必须为int; 变量的类型必须为int；赋值符号左边只能是某个变量或数组元素。

（6）return语句和函数返回值类型必须匹配。

（7）函数调用的参数个数必须和函数声明一致，且每个实参类型对应的形参类型必须相同。

**4.Mini C的目标代码和虚拟机**

Mini C的目标代码使用的指令集是TM机的指令集，因此如果代码生成的格式和Tiny语言的一样，那么就可以直接在TM机上运行。但是鉴于TM机是在命令行模式下运行，而Mini C虚拟机是可视化的桌面应用程序，所以需要进行一些修改。

**二、实验目的**

编译原理是一门综合性很强的课程，在理论学习部分，我们了解了编程语言的设计以及编译过程中的各个步骤。要更好地理解编译原理，动手实现编译程序是很好的方法。因此本次实验的目的是加深对整个编译过程的理解，同时提高软件设计以及编码的能力。

**三、实验文档**

**1. 词法分析程序的设计与实现**

词法分析过程基于以下DFA图（.vsd文件已放在doc/DFA文件夹下），实现过程参照了Tiny程序。



扫描程序对源代码逐字符扫描和处理，因此需要记录扫描到的位置，即行号、行内下标；对于标志符和数字串，则需要设置一个数组或者字符串进行记录，原则上标识符长度没有限制，但考虑到普遍的编程习惯中标识符不会太长，因此可以规定一个适合的最大长度。

对于单字符的单词，直接识别即可。对于多字符单词，则需要设置DFA的中间状态，尤其是含有公共前缀的单词，如“=”和“==”，当扫描到“=”时进入“INEQ”状态，表示该单词表示赋值符号或者判断相等符号，当下一个字符是“=”时才能确定当前单词是判断相等符号，否则是赋值符号。“>”和“>=”, “<”和“<=”等单词的情况类似。对于注释符“/\*\*/”的识别最为复杂，从DFA图中就可以看得出来。这里指出Tiny词法扫描程序的一个缺陷：当进入注释状态后扫到EOF不会给出提示，这个错误将在Mini\_C扫描程序中报告出来。

此外，对于标识符和数字的识别，Tiny程序使用的是标准库的isalpha(int ch)和isdigit(int ch)函数，这对于使用中文的用户不是很友善，若用户不小心使用了中文字符（注释除外），系统将会崩溃，因为这两个函数不能接受ascii之外的字符。在Mini C程序中改用了自己实现的函数。

最后，还有一处优化的地方——判断一个标志符是否为关键字，采用了哈希表进行查找，虽然Mini\_C只有6个关键字，但这里还是学习了大型编译器的做法。哈希函数和哈希表由参考资料中介绍的Gperf程序——完美哈希生成程序生成，完美哈希即为无冲突的哈希。生成程序的代码放在doc/gperf文件夹下。

**2. 语法树节点的设计**

在本编译程序中，几乎所有收集到的信息都存放在语法树中，后续的几个编译步骤都是围绕语法树进行。鉴于其重要性，在介绍语法分析程序的设计之前先介绍语法树节点的组成。

首先是子节点（child）和兄弟节点（sibling），是使语法树节点连接成一棵语法树的最基本的属性。

其次是节点类型（nodekind），使得每个节点具有其初步的特点，节点类型包括三大类：声明类型（deckind）、语句类型（stmtkind）和表达式类型（expkind）。声明类型包括普通变量声明、数组变量声明和函数声明；语句类型包括if语句、while语句、赋值语句、return语句、复合语句（compound）和函数调用语句；表达式类型包括标识符、常数和操作符。

对于标识符，要有其名字（name）；对于常数，要有其值（value）；对于符号，要记录操作符（op）；对于变量，要记录其类型（dataType）；对于函数，要记录其返回值类型（returnType）。为了报错及其他用途，也要记录它们的行号（lineno）。

上述属性是语法树节点较为基本的属性，给定一棵包含以上属性的语法树，可以还原出源代码。下面介绍用于语义分析和代码生成部分的属性。

在符号检查中，遇到声明节点就将它存入符号表，遇到标识符的使用则查找到相应的符号声明节点，然后“带”在身上（存在属性TreeNode \*declaration中），当想要知道一个标识符的信息时，可以通过其声明节点去获取。

在类型检查中，需要用到属性“表达式类型”，记录了标识符、常数、表达式、函数调用等节点的的类型。

生成代码部分需要的属性较多。对于变量，需要记录其是全局变量、函数形参还是局部变量，因为寻找它们地址的方式是不同的；变量和函数需要记录它们的偏移量（offset），函数的偏移量是指进入函数的指令序号；一个函数要记录它的局部变量存储空间的大小（一个int占用1个空间）和参数个数，以便于创建活动记录时分配空间；数组声明要记录其数组规模，在计算存储空间和偏移量时需要使用。

**3. 语法分析程序的设计与实现**

语法分析采用递归下降分析的方式，在这一过程中构建语法树。



文法2在语法上没什么特殊之处，但是这一步收集到的declaration全都是全局声明，因此可以在此处计算各全局变量的偏移量，以及全局变量的总大小（用于代码生成时分配存储空间）。





文法3具有左公因子 type-specifier ID, 而且二者都是需要进行记录的属性，因此扫描过程中需要对其记录，等到在出现分叉点后传递给文法var\_declaration或者fun\_declaration，因此这两个文法函数需要设计成带参数的函数。此外，可以在var\_declaration中计算局部变量的偏移量， 在fun\_declaration中汇总函数局部变量总的存储空间大小。



文法8去除左递归为

param\_list -> param (,param)\*

在此函数中，可以累计函数参数个数，最后在fun\_declaration中得到最后的结果。



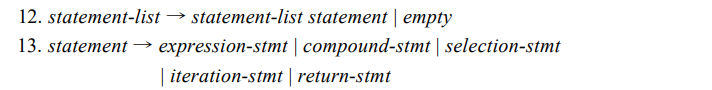
文法9具有左公因子，但这里右部不涉及其他文法规则，因此不用做特别处理。此外，在此函数中可以记录函数参数的偏移量。



文法11去除左递归为

local-declarations → var-declaration\*

其中var-declaration有左公因子，但和文法9类似，不用特别处理。其First集是{int, void}, 每个var-declaration若扫描到非first集的单词即可结束。



文法12也要去除左递归，类似文法11，其follow集是{‘}’}，当扫描到“}”时结束，其他则交给statement处理。而statement的First集是：

First = { ';', ID, NUM, '(', // exp-stmt

'{', // comp-stmt

if, // if-stmt

while, // while-stmt

return // return-stmt

}

扫到上述符号就分发给对应的statement函数，否则报错。对于if语句的二义性问题，实验指导里已经给出解决方案，这里不再赘述。其他语句没什么特殊之处。















文法18本身不复杂，但是对后面的文法影响很广，可以看见simple-expression经过一系列递归下降到factor后推导出var，所以expression存在左公因子var，而var又和call存在做公因子ID。

因此，在expression()中需要前探若干个单词，并把途中生成的子树做好记录，这里暂且称子树根部的指针为look\_ahead。首先前探一个ID，当在扫到左括号“（”时，即可判断当前语句是call语句，于是前探结束；否则当前的ID属于var文法，再进一步前探其是否是数组，随后结束var文法。此后若扫到赋值符号“=”，即可判断当前var属于赋值语句的左值，否则属于simple-expression。

前探结束后，若得到的是属于simple-expression的var或者call，就要将前面提及的look\_ahead子树根节点交给simple-expression函数，再一步步交到factor函数中，因此以下函数全部设计为带参函数：

TreeNode\* simple\_expression(TreeNode\* look\_ahead);

TreeNode\* additive\_expression(TreeNode\* look\_ahead);

TreeNode\* term(TreeNode\* look\_ahead);

TreeNode\* factor(TreeNode\* look\_ahead);

注意，look\_ahead并非总是要用上，当expression中扫到的不是var或者call，就没有需要传递的信息，赋值为NULL即可。

顺带提一下，additive\_expression和term都有左递归，因此都要改为以下形式或者类似形式：

term → factor factor\*



最后，arg-list也需要消除左递归，不再赘述。

递归下降函数的通用思路是：当遇到非终结符号时，进行函数调用；当遇到终结符号时，进行匹配。在匹配函数中，若当前扫到的单词和预期的一致则转入词法分析，获取一个单词后继续回到递归分析函数中，从而形成词法分析、语法分析交替前进的局面；当单词匹配失败时则报出错误，并回到语法分析。这样的设计模式不敢保证拥有很强的报错能力，因为有些错误对后面的代码影响很大，可能导致后面的语法分析失去其意义，这是一个缺点。但有一点可以保证的是，第一个报出的错误是真的错误。不管怎么样，只要出现一处错误，后面的编译过程都不应该继续下去，因此报错能力较低不能算作致命缺点，能够识别正确的代码就可以了。

**4. 语义分析程序的设计**

**（1）符号检查**

我称这一过程为“符号检查”而不是“建立符号表”，因为在本编译器中，符号表是为了检查符号是否正确使用，而不是收集一些信息，在这一过程结束后，符号表将仅剩下很少量的信息，而符号的信息都收集在语法树节点里了。

符号表用哈希表实现，采用拉链的方式解决冲突，哈希函数使用的是电子教材里介绍的哈希函数。拉链节点的属性有四个：标识符名称、标识符对应的语法树声明类型节点（TreeNode \*declaration）、作用域深度（scope\_depth）和下一节点指针。全局作用域深度为0，当进入函数、if语句、while语句时，作用域深度加1。哈希表支持插入、删除和查找操作。

符号检查的过程是对语法树的先序遍历，在此之前，不要忘记将系统函数input和ouput预先插入到符号表中。当遇到声明语句时，要检查符号表中是否存在同名且scope\_depth相等的节点，如果有则报出“标识符重定义”的错误，否则插入新节点，使用的是头插入的方法，因此链表中一个节点的scope\_depth值肯定大于或等于后一节点的值；当遇到标识符类型的节点（标识符的使用）时，要检查符号表中是否存在与该标识符同名的符号，如果没有则报出“未定义标识符”的错误，否则将查找到的第一个节点的declaration属性“交给”当前语法树节点，也就是为其收集信息，这里返回第一个节点是因为其scope\_depth最深，离当前标识符最接近。

当退出一个作用域时，应当把符号表中所有scope\_depth等于当前scope\_depth的节点删除，因为当前域的局部变量将不再有效了。这也避免了一个问题：如果不删除，那么符号表将很可能存在若干个同名且scope\_depth相同的标识符，无法进行分辨。显然，符号检查结束后，符号表中只剩下全局变量和函数了，这就是前面所说为什么不称这一过程为“建立符号表”的原因。此后，除了几处在代码生成过程中生成函数调用指令时需要查找函数声明节点外，符号表别无它用了。

最后还有一处需要检查的地方：最后一个声明是不是名为main的函数声明，因此要不断记录最后一个声明节点。

**（2）类型检查**

Mini C的类型比较少，只有void, int，数组和函数四种类型，因此类型检查过程比较简单。类型检查的过程是对语法树的后续遍历。当扫描到一个标识符类型的节点时，我们就可以知道其类型了，因为在符号检查已经将其声明信息挂载在该节点了。当遇到赋值语句和操作符类型的节点时，查看其左右两边的表达式是否都为int类型。函数调用语句的类型是其声明的返回值类型，此外还要检查参数个数和参数类型是否正确。对于return语句，其表达式的类型应与函数声明的返回值类型一致（如果为void, 返回语句应为return;或不存在）， 如果返回值类型为int而缺失return语句，则要报出“缺少return语句”的错误，实际上这个错误是在符号检查时报出的，因为符号检查过程中要为return语句记录其函数声明的信息。

**5.目标代码生成程序的设计**

TM机有8个寄存器，在生成代码之前，先约定哪个寄存器存放什么数据。数据寄存器有三个：0寄存器——存放常数0的寄存器；寄存器1和寄存器2存放操作数或者地址。3号寄存器没有用途。4号寄存器存放栈顶指针sp（stack pointer），总是指向栈顶位置。5号寄存器存放当前活动记录的基址bp（based pointer）。6号寄存器存放全局变量区域的基址gp（global pointer）。7号寄存器存放下一指令地址，即程序计数器pc（program counter）。

内存栈的结构示意图如下（假设栈大小为1024，下标变小的方向是栈顶方向）：

|  |  |  |
| --- | --- | --- |
| 栈内容 | 下标 | 备注 |
| 全局变量1 | 1023 | gp指向此处 |
| 全局变量2 | 1022 |  |
| 全局数组变量3（大小为10） | 1021 |  |
| …其他全局变量 | 1011 |  |

一次活动记录（函数调用），在栈中分配空间如下，假设一个函数为int func(int a, int b[], int c)，假设该区域下标从1000开始。

|  |  |  |
| --- | --- | --- |
| 栈内容 | 下标 | 备注 |
| 参数c | 1000 | 注意：偏移量越大的参数离bp越远 |
| 参数b | 999 |  |
| 参数a | 998 |  |
| return pc | 997 | 函数结束后跳转去的指令序号 |
| old bp | 996 | 存放主调函数的基址。寄存器bp指向此处 |
| 局部变量区域（假设大小为10） | 995 |  |
| 其他中间结果 | 985 |  |

目标代码生成的过程是对语法树的线序遍历，这是编译程序第四次也是最后一次和语法树交互。

当扫描到函数声明节点时，立马让函数的偏移量（offset）等于当前指令序号，也就是指明了函数的入口。然后将主调函数的bp（old bp）存好并更新bp为当前函数的bp，随后为局部变量分配空间——sp移动局部变量存储空间大小个单位即可。在生成函数主体（递归调用）之后，要考虑函数结束时要做的工作——弹出old bp和return pc以恢复主调函数的活动记录，实际上这一部分的工作应当放在return语句节点中完成，但考虑到void型返回值的函数未必存在return语句，因此对于返回值为void的函数结束工作在其声明节点中完成。而返回值为int的return语句，则在生成表达式之后再恢复主调函数的活动记录。

变量或数组变量的声明节点不需要生成代码。

下面介绍函数调用（call）节点的代码生成。在函数调用之前，先装载参数，需要注意的是，偏移量越大的参数应该先装入内存，根据活动记录的栈示意图就可以知道其原因。随后记录函数结束后应该跳转到的指令序号（return pc, 是函数跳入指令序号+1）。随后生成跳转指令——使pc指向函数的offset即可。最后，应该释放函数参数占用的空间——这条指令也就是return pc指向的地方。

下面分别介绍if语句和while语句的代码生成。

在if语句中，首先生成条件表达式的代码，随后要生成一条跳转指令——若判定结果为false则绕过then-part，然而目前还不知道要跳转到哪里，要等到知道的时候才进行“回填”，所以要记录下这条指令的序号loc1。随后生成then-part的代码，同样的，then-part结束后要生成一条跳出if语句的指令，但现在还不知道出口在哪，因此也要记录下当前指令序号loc2。注意，现在知道else-part的入口了，所以现在可以回填。随后生成else-part的代码（如果存在else-part的话），然后if语句的出口也知道了，可以进行回填。

while语句的道理也相似。首先需要记下条件表达式的起始位置，因为while-body结束后需要跳转到此处。在生成条件表达式后，要预留一条判定为false时跳出while语句的指令，但目前还不知道while语句的出口位置。while-body结束后，生成一条无条件跳转指令——跳转到条件表达式；此时while语句的出口位置也知道了，所以此时可以进行回填了。

赋值语句的生成比较简单，首先生成左部，将其地址存放在bx寄存器，然后生成右部，将其值存放到ax寄存器，最后将ax中的值赋给bx指向的内存地址即可。

对于常数的代码生成是最简单的，将其值存入ax即可。

对于变量的处理较为复杂。访问变量可能是取其地址，也可能是取其值，因此要设置标志用以区分。取值之前要取地址，所以访问变量时寻找其地址是必须的。根据上面的栈示意图可以知道，全局变量、局部变量和参数变量各有一块存储位置，所以寻址的方法各不相同。对于全局变量，其地址是gp – offset；对于参数变量，其地址是bp + offset + 2, 因为bp的位置和参数区域隔着一个return pc；对于局部变量， 其地址是bp – offset。取完地址后，如果还需要取值，对于普通变量，将地址中的值存入ax就可以了；而对于数组元素，还要得到其下标值，然后计算数组元素的地址再进行取值。

最后讲讲运算表达式的处理。无论是算术表达式还是条件表达式，都要求得操作符两边表达式的值，存放在寄存器ax和bx里。对于算术表达式，只需要将两个寄存器里的数值进行相应的计算后存放在ax里即可。对于条件表达式，将两个值进行减运算（SUB）后存放到ax里，然后根据相应的操作符进行判定，如果为true则将ax设置为1，反之设置为0，随后无论是if语句还是while语句，在进行false无条件转移时（该转移指令是JEQ指令），如果ax里的值为0则说明要跳转，否则不需要跳转。

以上是遍历语法树过程中对一些节点的处理。除了扫描语法树过程中生成的代码之外，还需要额外做一些工作。例如一些初始化工作：设置gp指向栈底（TM机将所有寄存器初始化为0）；为全局变量分配存储空间（在生成语法树时，用一个变量global\_size记录全局变量的总大小，注意，这个变量不是任何语法树节点的属性），也就是将sp偏移若干个单位；生成input和output函数的代码。在语法树部分的代码生成完毕之后，生成一组call指令以调用main函数（main函数的主体部分在遍历过程中已经生成），从而让程序开始执行。

**6.虚拟机程序的改造**

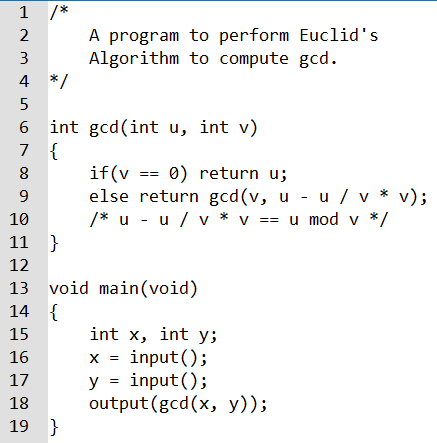
Mini C虚拟机仅在TM机基础上做了略微的改造，主要包括两个方面：①对输入/输出指令的改造：Mini C的控制台是桌面应用的模式，而TM机的控制台是命令行模式，因此输入输出的方式不一样，因此将IN指令改为弹出输入对话框的方式，接受用户输入的整形数，而OUT指令则是将计算结果显示到“运行结果”文本框中。

②Mini C虚拟机只有“执行程序”这个步骤，即按顺序执行所有读取到的指令，相当于TM机中的’g’命令。

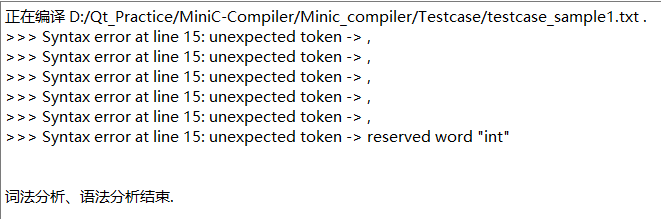
**7.测试**

所有测试用例已经放在exe/Testcase文件夹中，语法树文件和目标代码文件在编译过程中自动生成，文件名以源代码文件名为前缀，分别以\_SYNTAXTREE.txt和\_CODE.txt为后缀。测试用例涵盖对词法、语法以及语义的测试，最后还有项目设计指导书中给出的两个测试用例，在此也仅展示这两个测试用例的测试结果。

**（1）testcase\_sample1.txt**



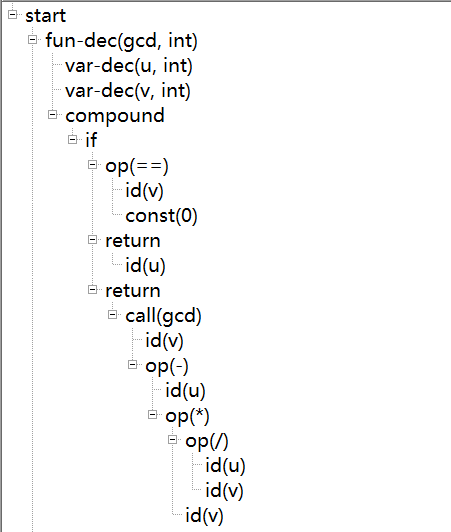
编译结果：

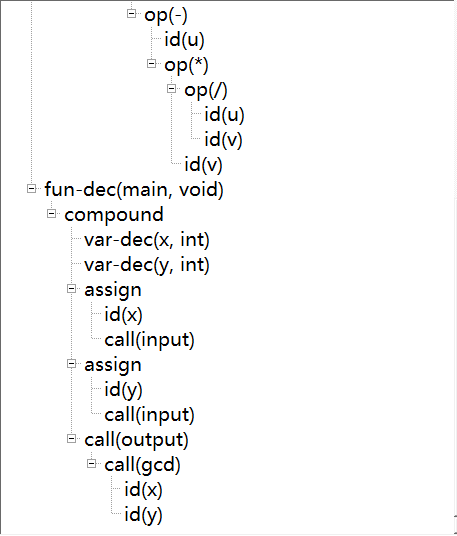


可以发现第15行语法错误，应改为

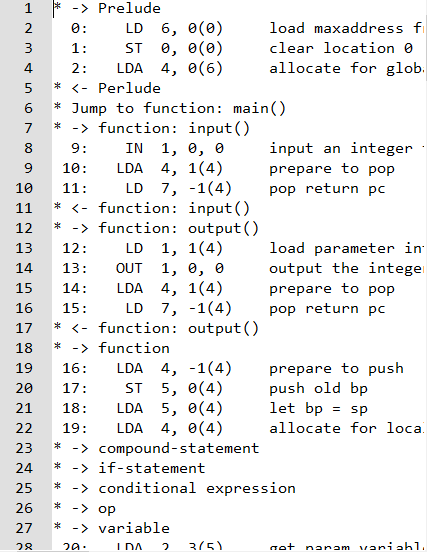
int x; int y;

更改后编译通过，生成的语法树为：

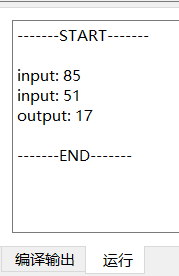




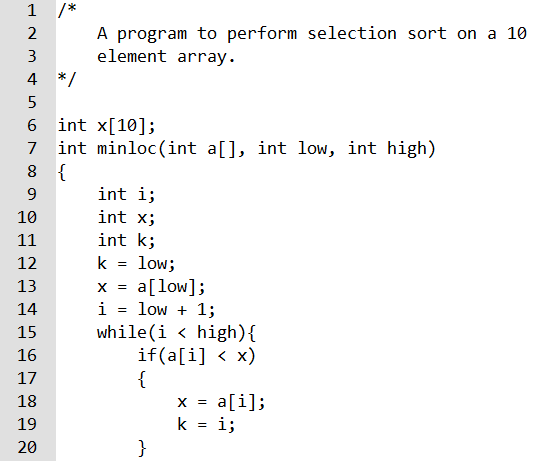
代码生成128条指令，具体结果可以参见testcase\_sample1\_CODE.txt文件，部分截图如下：

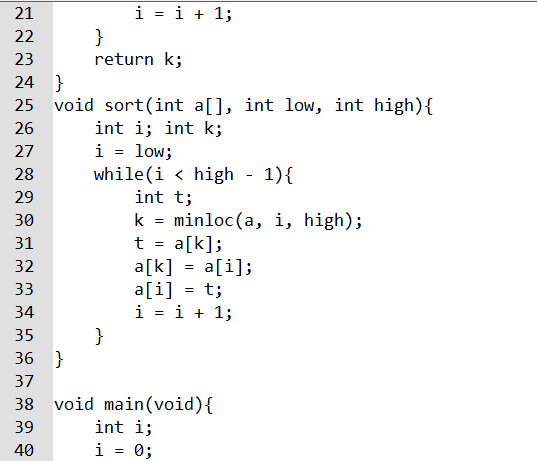


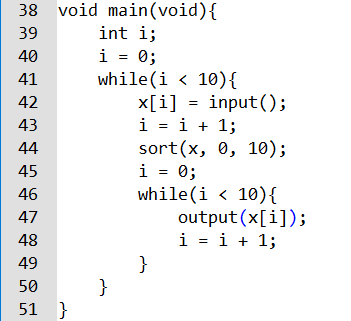
运行结果如下：



**（2）testcase\_sample2.txt**

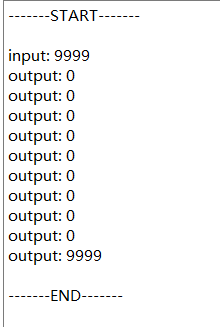






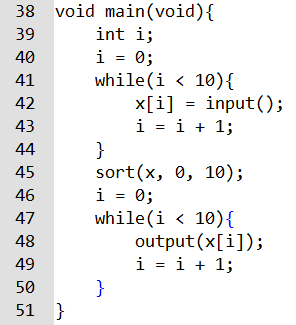
编译通过，语法树和生成代码太大了，因此不做展示。

运行结果如下：

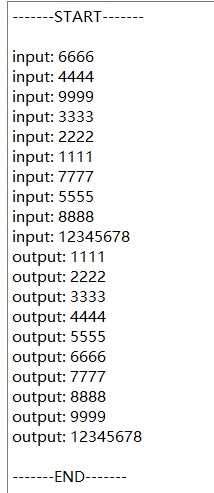


程序只要求输入了一次整数，输入9999得到以上结果，查看main函数的代码发现第一层while的有括号放错位置了，但为什么数组的其他9个元素未初始化也能使用？因为虚拟机程序将内存栈的每个单元初始化为0，而为数组分配空间时仅仅是移进了栈指针，所以根据数组首地址和偏移量寻找某个元素时，如果它从来未被更改，那么得到的值是0。

main函数更改如下：



运行结果如下：

排序正确。

**四、实验总结（心得体会）**

陈栩明同学的心得体会（单人小组）：

历经了一个多月的时间，陆陆续续地完成了Mini C编译程序，回头看看觉得这个过程充满了坎坷。此时心中没有太多的想法，诚实地说，最真切的感受就是——仿佛卸下了一块心中的大石。此外还有一丝丝失落——这个Mini C太“迷你”了，以至于一点也不实用，但它是实实在在地花了许多功夫才完成的。我很佩服那些写大型编译器的技术人员，更佩服创造第一款高级编程语言的Backus，他们花费的心血不可计量，他们遇到的困难我不敢想象。这么一想，我突然觉得我所做出的努力简直微不足道。我们可以偶尔抱怨我们面对的困难，但不应该以此为常态，在大部分时间里，还是应该以端正的态度积极地克服困难。

**五、参考文献**

1.《编译原理与实践》电子书（语义分析、运行时环境、目标代码生成）

2. Tiny程序源码

3. Tiny虚拟机源码

4. Qt5 官方帮助文档

5. C++ Reference

6. 一系列博客文章以及论坛里的技术讨论

7. Github项目<https://github.com/benfowler/c-minus>（语法分析部分）

8.Github项目<https://github.com/koyabr/C-Compiler>（代码生成部分）

**六、项目以及各成员自评**

**1.项目完成情况的自评分数以及原因说明**

项目完成情况：按时完成。

自评分数：95/100

原因说明：

按时完成实验内容，功能完善，文档比较齐全，编码风格良好，界面虽然不算美观但也比较简洁，软件易使用。

**2.各成员自评分数以及.原因说明（包含本人需要完成的分工情况，实际的完成或付出情况以及实际工作量或付出在整个小组完成总量中的占比情况说明）**

陈栩明同学自评分数：95/100

原因说明：

个人完成所有的工作，对编译原理实践这门课程比较热心且能认真对待，对于实践过程中遇到的难题，能够上网查找资料或与同学讨论，坚持不懈地解决问题。