编译实验难点分析

15061062 缪佳敏

一、词法分析

词法分析这部分比较容易，刚开始写的时候根据书上的算法就能很清晰容易地写出来。其中有几点需要注意：

1. 字母包含下划线\_

2. 整数不能有前0（这部分我容错处理了）

3. 字符和字符串存储的时候要加上 ’ ’和 “ ” 一起，这样后期处理比较方便。

二、语法分析

语法分析部分书上也有部分代码示例，我就模仿书上的代码写完了这部分内容，逻辑也是比较清楚的，不是很难，但是代码量很大。

这部分我遇到的问题是读到某些单词之后需要进行回溯，但是我词法分析的时候采用的是infile.get()函数，文件指针回指不是很方便，一开始是准备改词法分析那边的读文件的方式，后来思考了一下，有回溯的只有很少的几种情况，以int /char 标识符 开头的可能是变量定义，也可能是函数；以标识符开头的可能是单个的标识符，也有可能是数组或者函数调用。因为我的词法分析str只能存储当前的单词，前面的会被覆盖掉，这就出现了想要回溯的时候无法找到之前内容的情况。于是，我就开了一个小型的string 类型的数组，一旦遇到int 或者char 或者标识符，我就将它们首先存进string数组中，这样在回溯的时候，首先直接在数组中寻找就可以了。

三、符号表管理

符号表管理这部分我写得比较简单，采用了一个无序表来存储。符号表的数据结构包括了名字，种类，类型，常量的值，参数个数或者数组大小，层数，在函数中的相对位置。

这边我一共遇到了三个问题：

1. 查找符号表的时候如何区分两个相同名字的变量或者常量是重复命名，不允许存在；还是在两个不同的函数中，可以存在。（这部分的解决方法是将函数名也存储到符号表中，查表的时候从后往前查，在遇到第一个函数名，即该变量或常量所属的函数前，若发现有相同的名字，那就是重复命名。如果是在遇到的第一个函数名之后发现有相同的名字，那就是属于不同的函数，这种情况可以存在，将常量或者变量加入符号表。）

2. 刚开始的时候，我没有写在函数中的相对位置，直到生成目标代码的时候发现需要使用到在函数中的相对位置，于是从头开始，给每个加入符号表的常量或者变量又加上了相对位置，思路很简单，但是要修改的地方非常多，很麻烦。

3. 生成四元式的过程中会产生一些临时变量，这些临时变量在生成目标代码的时候也要被使用，于是就在生成四元式的过程中又往符号表中插入了一些临时变量。

四、语义分析和四元式生成

这部分真的是有点难了，刚开始的时候不知道怎么下手，开始写之后又遇到了很多问题，后来又因为没有具体记录每种四元式的存储内容代表什么，在生成目标代码的时候带来了极大的不便。

就四元式生成而言，我认为以下两个问题还是值得注意的：

1. 表达式的四元式生成

生成四元式的过程中很多地方都涉及到表达式，表达式部分的四元式生成是一个很重要的难点。表达式还涉及到项和因子，首先是因子部分，因子包含多种情况，如果有数组，直接先生成一个使用数组的四元式，然后将存储数组的中间变量返回，传递到项中去；同理，函数调用的情况，将中间变量返回；如果是整数，要将负数特殊处理一下，也要先生成一个四元式处理负数，然后将中间变量返回；如果是标识符，就直接将标识符返回给项。项利用因子传递过来的返回值，生成含有乘除运算符的四元式，并且将存储结果的中间变量返回给表达式。同理，表达式利用项传递过来的返回值，生成含有加减运算符的四元式，并且最终返回一个中间变量。

2. 另一个比较难的地方是label的设置

在if、do-while、switch这类语句中会涉及到跳转语句，跳转位置就需要设置标签来标记了。

在处理if语句的时候，已进入if语句，就生成一个新的label，存储在一个全局的string label里面，然后将if的条件生成四元式，然后生成BZ label这条四元式，它的含义是，不满足就跳转。在处理完if之后，将刚刚生成的label设置在这里。这整个的含义是，进入if之后先判断是否满足条件，如果不满足，直接跳转到if结束后面的label，不执行if内部的四元式。

但是，如果出现多层循环嵌套就会出现label混乱的情况，例如：在一个if里面嵌套另一个do-while，生成四元式的过程是这样的，首先在进入外层if的时候生成label1，此时label = label1；然后在进入内层do-while的时候生成label2，此时label = label2，在此处设置label；结束内层do-while的时候，BZ label，此时label = label2；结束外层if的时候设置外层label，但是，此时的全局变量label已经被改变了，不是label1了，这就会出现label混乱的情况。

我采用的处理方法是，将全局变量label改成局部变量，然后在函数中添加一个参数，用来传递label，这样就避免了刚刚描述的那种情况了。

五、目标代码生成

目标代码生成这部分就很难了，主要是运行栈的设计和数组的处理。

代码运行最开始的时候，将$fp = $sp，然后根据符号表中的顺序存储局部常量，局部变量和临时变量，在存储之前，需要为这些常量或者变量在运行栈中开出一段空间（通过$sp向下移动来实现开空间，通过符号表中存储的相对位置和与$fp之间的偏移量来找到应该存储的位置）。遇到函数调用的时候，首先记录此处的$sp，然后再给调用函数的参数开一段内存，接着存储之前记下的$sp，$fp，$ra。此时再将$fp = $sp，首先存储一下该函数的参数，然后与之前过程相同存储局部常量变量等等，如果改函数调用内部还有函数调用。在函数执行结束之后，跳转到$ra所指的指令，恢复$sp， $fp，继续执行下面的指令。需要注意的是，函数执行的结果存放在$v0里面，该寄存器不能用作其他功能。

另一个需要注意的就是数组了，第一数组在进行存储的时候不能像其他常量或者变量一样开空间的时候之间-4，而是要计算数组的大小，然后在修改$sp的值。第二，在进行数组赋值或者数组使用的时候，需要通过下标计算具体的存储位置，需要注意的是，如果数组是全局变量（即存在.data里面），注意随着下标的增加，在内存中的位置是不断增大的；而如果是局部变量（即存在运行栈中），随着下标的增加，在内存中的位置是不短减小的。

六、错误处理

错误处理这一部分，我认为没有什么难点，只要注意一下处理完错误之后应该读到哪个有效字符就可以了。但是，错误处理涉及语法错误和语义错误这两部分，其中语法错误有很多种情况，非常繁琐，需要花大量的时间来写。

七、代码优化

1. 消除公共子表达式

关于消除基本块内公共子表达式，书上给的算法是先划分基本块，然后再利用DAG图消除局部公共子表达式，我认为对于我们目前写的编译器其实有更加简单的做法。

首先，对于基本块，其实是可以不用划分的，通过大量四元式的分析，我发现其实能够划分在同一个基本块中并且有公共子表达可以消除的，只有和+ - \* / = 有关的四元式，而且这些四元式一定是连在一起的。所以我们可以直接用两个循环来实现和划分基本块一样的功能。其次，对于消除公共子表达式，我们可以通过操作符一样，两个源操作数一样来确定这是两个相同的子表达式，值得注意的是两个源操作数不能被二次修改，并且这些判断是在同一个函数内部进行的。

找到相同的子表达式之后，将对应的使用的目标变量替换，并且删除该公共子表达式。

2.活跃变量分析

逆向遍历四元式，用数组来存储活跃变量，用邻接矩阵来存储各活跃变量之间的联系。

根据use和def来增减活跃变量数组中的值，每加进去一个活跃变量，就将它与数组中的其他活跃变量连接起来（在邻接矩阵中找到对应的下标，将元素置1，表示有连接）。要注意循环和分支的情况，循环和分支可能会删掉一些不该删除的活跃变量，所以，可以将遇到的循环和分支中的活跃变量单独处理，存在一个单独的数组中，然后将该数组中的值加进之前的数组中。

最后得到的邻接矩阵就是一个冲突图了，graph[i][j] = 1，表示活跃变量数组中的i和j下标所对应的活跃变量冲突。

3.全局寄存器分配（着色算法）

该算法参照课本376页的算法15.1，此处不再赘述。实现起来还是比较简单的。

总之，冲突图的构建是优化部分最难的内容了。