编译原理课设申优文档

17373132 赵久昂

**前言**

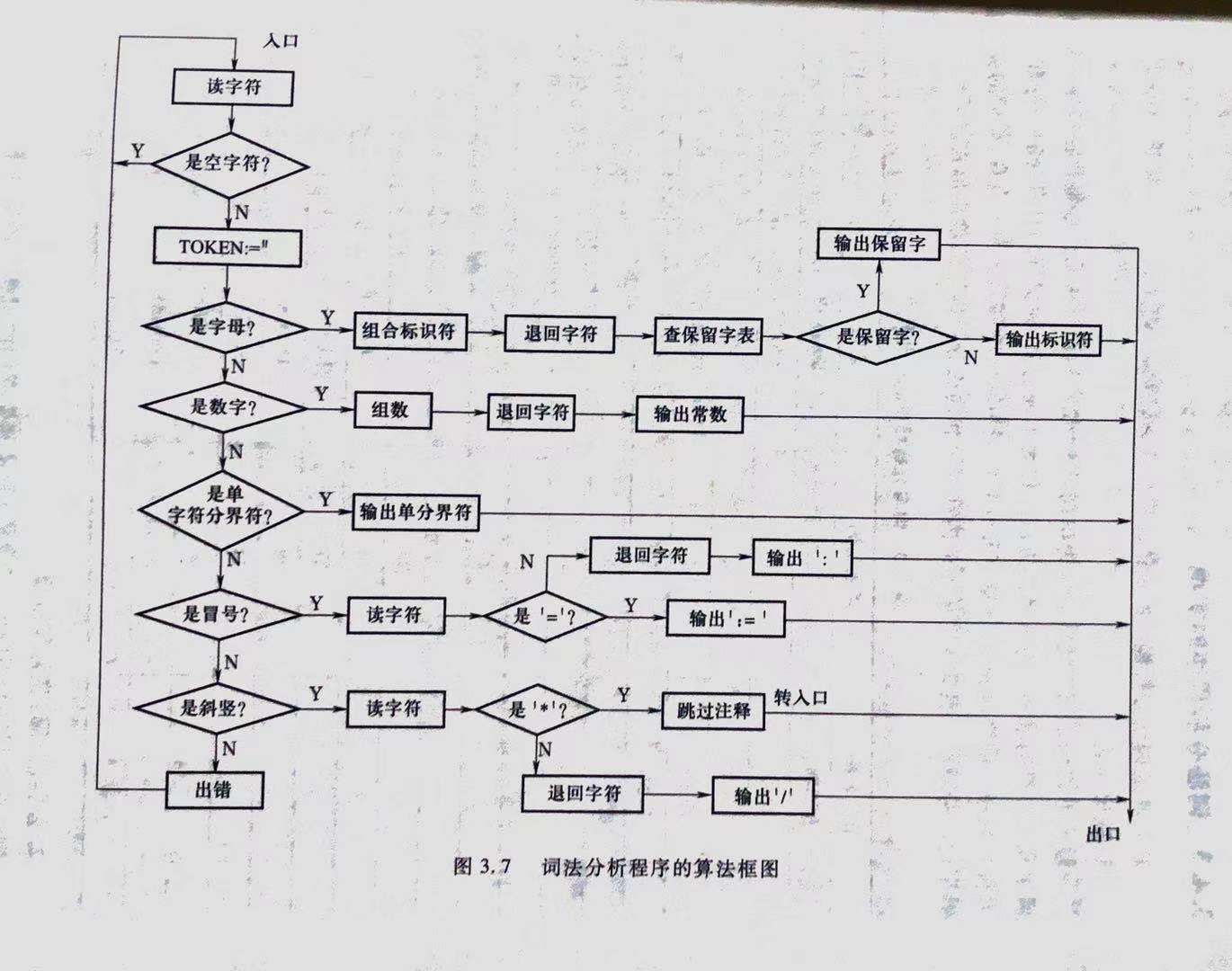
编译原理课设任务持续了三个月，已经接近尾声。在三个月的编译器设计中，课程难度越来越大，但是对编译器层次化以及循序渐进的设计，让整个设计的工作量减少了不少。在整个编译器的设计中，由底层开始，先后进行**词法分析、语法分析、错误处理、代码生成、代码优化**的五项任务。其中，词法分析是最基础、最简单的部分，代码优化是最困难的部分。在本文中，会先后对五个小任务分别进行说明。

## 一、词法分析

词法分析的作用是扫描源程序的字符，按语言的词法规则识别出各类单词和符号，并将相关的字符的组合作为一个整体单词来输出，比如一个程序语言的保留字、标识符、常数、运算符等。词法分析的主要工作有：

1. 一个一个字符的读入，存到一个临时的字符数组里，然后通过判断字符数组的内容来识别出各类单词。有的时候还需要超前读一个字符才能确定是什么单词符号。
2. 删掉多余的空格、换行符、制表符。

在词法分析后，我们可以将分析出的单词表存起来，便于下一步的语法分析使用。但是我在设计的过程中没有考虑到这一点，语法分析和词法分析进行同步执行，为后续的设计带来了一些麻烦。词法分析的算法框图如下，具体的为代码实现可以参考《编译技术》教材P71-P73。

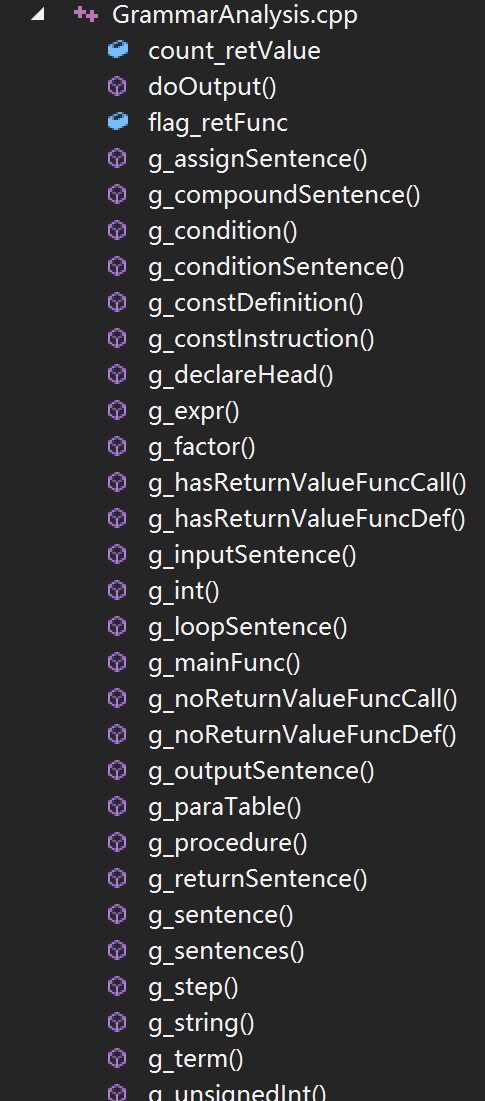


在我的程序设计中，词法分析是作为了语法分析的子程序，并不是先词法分析，将单词结构化存储后再调用语法分析。而是当语法分析程序需要读进来一个单词的时候，调用词法分析程序，并将调用的结果存到全局变量中，这样语法分析程序可以间接接收到词法分析程序的分析结果，进行相应操作。

## 二、语法分析

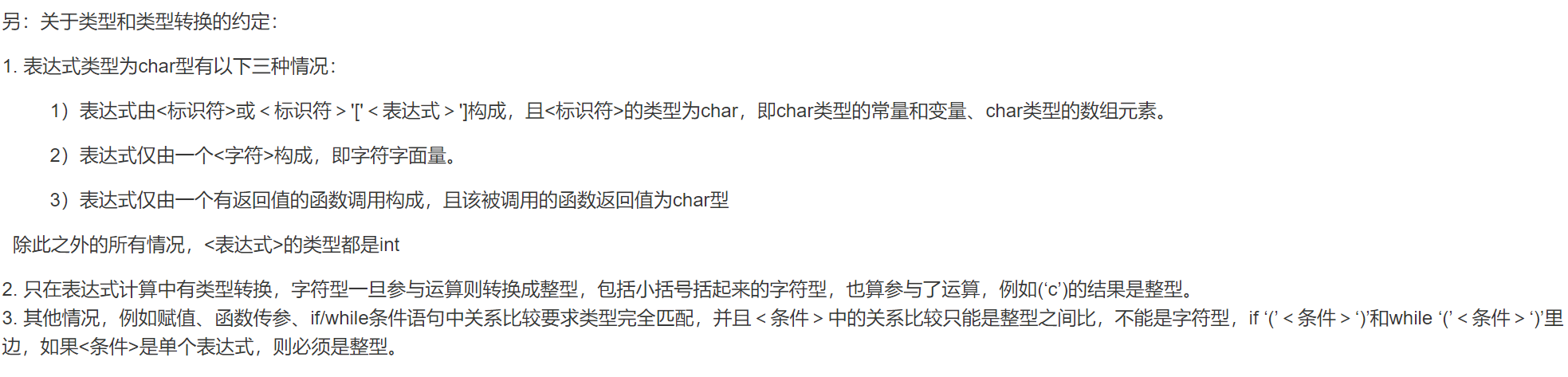
语法分析是编译过程的核心部分。它的作用是按照文法规则，从源程序的符号串中识别出各类语法成分，为下一步代码生成做准备。

在我的程序设计中，我采用的方法是带回溯的自顶向下分析方法。但其实我也建议采用改写文法，将需要进行递归分析的文法规则进行相应的修改，这样可以大大提高效率。如果采用带回溯的语法分析，我个人认为可以让语法分析程序的结构更具有层次化和逻辑性，每一个函数只需要判断一条文法规则（非终结符推出……），并且如果文法有了相应的改动，只需要修改相应的文法判断函数即可，而不会有“牵一发而动全身”的复杂情况出现。大概的程序设计结构如下：



需要注意的是回溯问题。由于我的语法分析程序并不能自动做到回溯效果，即：如果对于一条语句，进入了错误的语法分析函数，那么判断不匹配时跳出函数的时候，当前所读到的单词和各种符号标志位都不是进入函数前的状态，这样的话算是没有处理回溯问题。我的做法是：**在进入每一个函数的时候，都进行一个“保存现场”的操作**，将这些会在语法分析过程中修改的变量进行一个压栈操作。如果碰到了不匹配的情况，则进行“**恢复现场**”的操作，将刚才压入栈里的变量都取出来复原，这样就可以保证回溯后，用其他规则判断的时候的语法分析的状态时正确的。其实这相当于我们手动维护了C语言函数调用的一个状态，如果想避免这个操作，可以将每一个语法分析函数的返回值进行结构化处理，但是相应的也会消耗大量时间和空间复杂度。

在后续编译器的设计过程中，我们发现会有判断表达式类型是int还是char的情况（对于相应的规则，课程组已经明确提出要求）。



我们语法分析的过程中是要生成相应的符号表的，在生成符号表的时候，我们就需要对相应的变量进行一个标志（标志是int/char，或者标记是否为数组还是常量）。那么我们怎么来区分这个变量的值是int还是char呢？在判断表达式、项和因子的语法函数中就要进行相应的处理——如果返回的表达式是int，则相应的语法分析函数的就返回return 1，若为char则return 2。根据规则，表达式类型为char的三种情况很容易判断，因此在判断的时候需要注意两点：参与运算因子、项的类型和个数。如果个数大于1，则说明最终的结果肯定不是char，因为char不可能由两个因子运算后的结果构成；如果个数等于1，则在判断其类型是否为char。在复杂表达式的分析中，我们语法分析的程序关于表达式、项和因子的处理分析，往往会出现一个递归调用然后返回的情况，因此通过这种判断语法分析函数的返回值的方法，利用了语法分析程序的递归性质，可以很好的判断出表达式的类型。

**符号表是一个十分重要的结构**，它存储了每一个标识符的相应的信息：如char/int，是否为数组、是否为常量等等。在语法分析的过程中，需要常常对符号表进行访问、添加。我们知道，局部变量的存在的有效范围只在该函数里，所以符号表的构造也需要处理重名的情况。

我的处理方法是：首先整体上是有一个函数的符号表，其中存放的member是函数名、函数返回值类型、函数参数个数及类型、函数内的变量的符号表。在处理一个函数的时候，我们先建立一个函数的符号表项，在语法分析的过程中相应的修改这个函数表项中的内容，同时，我们也要新建一个变量的符号表，每定义一个变量的时候要根据定义的具体信息来将信息添加到这个变量的符号表中，最后再将变量的符号表move到函数符号表项的member变量符号表。

## 三、错误处理

错误处理相对而言也是一个比较简单的部分。因为错误的种类有限，只需要对每一种错误，在程序中找到其可能出错的位置，进行相应的判断即可。稍有难度的是判断**函数传参时的传参类型**是否匹配，这个由于我们在前文符号表的构建中提到了，我们将一个函数的参数都存到了这个函数表项的member中，因此我们在调用函数的时候，我们先将传进的值参数表取出，然后与函数的参数表进行一一类型比较，以及个数的比较，如果有不同则输出错误信息。

## 四、代码生成

代码生成分为两步。

1. 从源程序到中间代码
2. 从中间代码到mips

我的程序中，中间代码是一个类imCode，有多个类继承这个父类，分别代表这个中间代码的类是哪一种（函数调用/表达式/条件判断）。

从源程序到中间代码是相对简单的一步。对于源程序中的每一个语句，相应的将其修改成四元式，存到中间代码的结构里，并给类中的member添加上相应的type，便于后续查找分析。

其中比较难的是表达式的翻译，对于一个复杂的表达式，比如x = a \* (b+c\*d-(f / 9 + 3))。我利用了语法分析的递归下降的思路，在语法分析程序判断表达式语句的时候，我们实际上是不断下降，直到分析到了表达式的最内层，在本例子中是 f / 9，也就是说，语法分析表达式的顺序，实际上是从内到外的，也就是符合我们的正常的运算顺序。因此我们可以在语法分析的程序中就new一个imCode的对象并且对其相应的四元式的每个位置进行赋值。需要注意的是，我们这个时候还需要分配一个临时变量，临时变量的分配可以通过一个计数器来实现。比如T1 = f / 9，那么再申请一个新的临时变量，应该为T2。

在生成表达式类型的四元式的时候，我是进行了一个栈的处理，每次读到一个因子就压进栈中，然后读进符号可以运算的时候，pop出栈顶的两个元素并进行运算，然后将T1再压回栈中，便于下一步的四元式生成运算。有了这个栈的结构，生成表达式、赋值语句、条件跳转语句等，都会变得十分方便。

对于每一条中间代码，应按照顺序存到vector容器中，便于下一步翻译成mips指令。

从中间代码到mips稍微有些复杂，需要判断每一条中间代码的类型，并作相应的处理。由于是第一次代码生成作业，因此不需要分配寄存器，于是就相应的把所有的变量值都存到内存中去，建立一个映射得关系。取得时候使用lw指令，用完之后就用sw指令存回内存中。

在这个时候我们就会发现，如何能在准确的存取呢？实际上要在语法分析构造符号表的时候进行修改，每读到一个变量，建立一个新的符号表项的时候，都要对其**进行一个内存的偏移量offset的分配**，固定这个变量。如果想存取的话，只能使用当前函数的栈中偏移offset大小的内存空间。在分配完这个函数内的变量的空间（包括临时变量）之后，我们其实已经就确定了这个函数的栈的大小了，我们再把叠加起来的offset存到函数的函数表项里来确定这个函数要用多大的空间，这样的话，每次跳转到一个函数之后，先使用subi $sp $sp offset指令对当前的栈进行一个更新，然后返回语句的时候在addi $sp $sp offset，将栈指针挪回调用函数前的位置。这样就可以很好的对每一个函数的空间进行维护。

值得注意的是，对于函数传参的处理，在本次最初的代码生成就可以相应的分配$a1,$a2,$a3三个参数的寄存器，但是如果在调用的子函数中产生了跳转，要相应的保护现场，并在跳转回来后恢复现场；在有返回值的函数调用的时候，函数的返回值可以存给$v0。

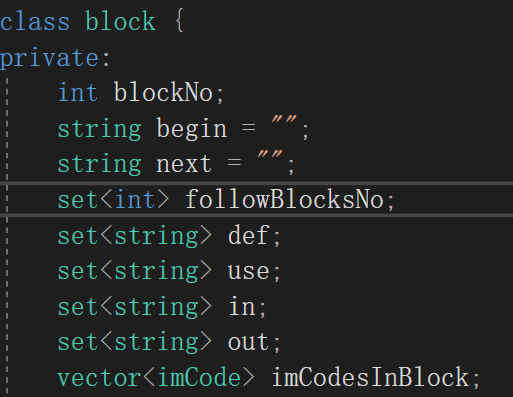
## 五、代码优化

代码优化是最困难也是最容易出错的地方。在代码优化的时候，我们需要做的最重要的一件事情，就是给变量分配相应的寄存器。分配全局寄存器的方法有：引用计数法，利用活跃变量分析构建冲突图法。我选择了了后者。对于临时寄存器的分配，我采用了临时寄存器池来进行动态分配。

采用冲突图法分配全局寄存器方法如下：

第一步，是划分基本块。划分基本块的判断条件是：

1. 整个语句序列的第一句是入口语句。
2. 任何能跳转到的第一句语句是入口语句。
3. 紧接着跳转语句的下一句是入口语句。



第二步，是构建流图，相应的后继关系要找准确。在这步中，我是在构建基本块的时候给每一个基本块的begin和next成员做一个标注，begin的意思是基本块是从哪个label开始的，end的意思是这个基本块下一步要跳转去哪里。当然，begin和end有可能为空，这样的话就说明这个块是顺序执行下来的，没有跳转。

第三步，是进行活跃变量分析，计算in，out集合。首先利用教材中计算use和def的方法计算出来每一个块的use和def。然后用相应的算法对块从后往前遍历，计算每一个块的in和out，并不断循环更新，直到每一个块的in和out集合都不改变，则停止遍历。

第四步，是构建冲突图，构建冲突图的算法是从1606学长马振亚的参考文档中借鉴的。具体算法如下:

**对于基本块B，首先令live = out[B]**

**然后对于B中的指令I，从后往前，开始遍历**

**对于任意变量x属于def(I)，对于任意变量y属于live**

**冲突图添加边(x,y)**

**然后更新live = use(I) U (live-def(I))**

第五步，进行染色。染色算法也是教材中的，s寄存器有8个，于是就首先遍历冲突图寻找一个节点的边小于8的点，将这个节点和对应的边删除。然后再不断遍历，如果冲突图为空，则算法终止；否则，将剩下的冲突图里的边多于8个的节点中的一个，标为“不分配寄存器”，然后删去，继续从头执行算法。染色的结果是每一个变量会被标注是否分配寄存器，以及相应的应该分配几号寄存器。由于染色算法是基于冲突图、活跃变量分析的，因此在分配寄存器的时候，可以直接分配，不需要做写回内存等操作（**部分特殊的递归程序除外，比如递归调用语句的后续还use了递归语句之前的变量**）。

第六步，重新生成新的中间代码。这一步重新生成新中间代码，是为了给相应的变量分配寄存器，这样的话，在中间代码至mips的翻译中，不需要考虑寄存器的使用情况了，在中间代码这一步就将寄存器分配好。由上文所述，我们已经分配好了全局寄存器，那么在每一个块中我们需要给临时变量分配临时寄存器，我才用的方法是FIFO分配，由于临时变量的存活时间很短，以及不跨块的特点（**函数调用除外，函数调用时的值参数如果有临时变量，则其不应该分配寄存器，否则会出现覆盖的问题，当然也可以通过替换t寄存器将t写回内存的方法来避免**），于是在进入一个新的块的时候不需要对t寄存器进行写回保存，也不需要保护、恢复现场。

其余优化操作：

1. 将for语句和while语句进行优化，正常翻译的话，会在for语句的开始进行branch的判断，在for的结尾会进行无条件的跳转至for语句开头的label。优化后相当于在进入for、while之前外加一条if来减少结尾不必要的j指令。
2. 赋值操作有的时候会跟在表达式之后，比如add $s0 $s1 $s2后接着move $v0 $s0来函数return，此时可以将两条语句合并成一条语句。
3. 临时变量的复用。在一个复杂的表达式中，我们往往会分配了很多新的临时变量，但其实并没有必要，完全可以将两个临时变量的结果重新存回其中的一个临时变量，这样可以大大减少临时寄存器的分配数目。
4. 无用代码的删除。进行了活跃变量分析和冲突图的构建，这一步与其算法几乎相同。如果一个语句的def在之后没有被use（live集合中不存在），就可以将其删除掉。但要注意，scanf语句由于与输入有关，所以不能删除。

**总结**

总体来说，编译理论课程设计相比其他专业课的难度要大，它不仅仅是一个代码结构设计的问题，还需要考虑编译器底层的实现原理方法。在整体的编译器设计中，我的收获不仅仅在于了解了编译器的原理，也提升了我的C语言编程水平，可以熟练使用STL中的容器和Class类的操作；在设计编译器的过程中，我也有思考如何将其设计的更具有层次化，更容易扩展。在设计的过程中，我碰到了各种各样的困难，无论是原理上的还是程序设计上的，但是大多可以通过阅读教材和上网查阅来解决。编译器的设计上有一部分算法并不是很完备的，需要我们独立思考和调整。作为一个高年级学生，我们更应该培养自己的创新能力，课程组刚好也提供给我们这样的训练，提升了自己的设计能力，为我们以后的科研和工作打下了良好的基础。