《编译技术》课程设计

申优文档

学号：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

年 月 日

## 一．课程设计的流程

课程设计的编译器的设计主要分为以下几个部分：词法分析、语法分析、语义分析以及中间代码生成、代码优化以及目标代码生成，中间贯穿着符号表的管理以及出错处理。其中词法分析比较简单，不断地读取单词供语法分析程序使用即可。重点在于语法分析，依据抽取的文法编写递归下降分析程序，处理各个语句成分，在语法分析过程中伴随着语义上的检查，此时就要涉及到符号表的设计，一个好的符号表意味着用较少的空间来存储更多有意义的信息以及能够用较少的时间来进行相关信息的查询获取。中间代码生成是一个从语法分析出的结果转化到目标代码的桥梁，设计好四元式的格式以及保证覆盖完全语法成分即可，在语法分析得到相应的语法成分后向四元式表插入相关的四元式便可完成中间代码的生成，难度并不是很大。再之后是目标代码的生成，相比于提高目标代码的效率性，目标代码运行的正确性更为重要，先保证目标程序的正确性才是首要任务。目标代码生成涉及到运行栈的设计以及空间分配、对于MIPS体系的理解以及对于程序运行的控制，特别是运行栈的设计，运行栈与程序运行的正确性有着极大的关系。下一部分是代码优化，代码优化依据书中的算法来实现，涉及到DAG的优化，寄存器的分配，常数合并以及冗余代码的删除等。整个设计流程中还涉及到出错处理，为不同的错误定义相应的错误类别以及解决方案，遇到相应错误则传入错误，而后调用相应的解决方案即可。

## 二．课程设计中遇到的困难以及解决方案

**1、词法分析阶段：**

词法分析比较简单，需要注意的是与之有关的错误如何处理。比如遇到字符或者字符串不符合文法定义时，要优先读到下一个双引号或者单引号所在位置，找不到下一个双引号或者单引号，则需要处理到该行结束；在遇到读入字符不是文法的合法字符时，需要递归调用insymbol()方法去获取单词，直到获取到合法单词为止。

**2、语法语义分析阶段：**

一、第一个问题便是符号表项究竟要包含哪些属性，刚开始的时候，我设计的符号表项存储了name，kind, value, arraySize, address五个字段，分别代表标识符名字、类别、数值、数组大小以及地址，但是之后处理函数调用的时候，发现要进行函数的检查工作，所以又在符号表项中加入了paraNum这一项，代表函数的参数个数，这样符号表项的设计就算是完全了，可以满足语法以及语义分析的要求了。symbolTable[]为符号表项的存储集合，其中存放了程序中的所有数据，先记录全局数据，而后以函数为单位进行处理，每处理完一个函数后，则将该函数除了函数名字以及函数参数外的其余数据进行冲表删除

二、除了符号表，要想在查找数据的时候速度较快，我又设计了一个分程序索引表displayTable[]，分程序索引数组存储每个函数在符号表中的地址，通过该分程序索引表即可快速查找到与该函数有关数据信息的地址。这样在查找某个函数中的数据时，先通过分程序索引表找到该函数在符号表中的开始位置，再去查找相关数据，会免去许多不必要的查找。

三、在处理表达式这一部分的时候，因为要涉及到数据类型的检查，故我用一个全局变量factorType记录因子数据类型，每次调用factor()处理时，便将factorType赋值为处理完的数据的类型，这样在最终表达式处理结束后，factorType记录的是该表达式的数据类型，通过判断factorType即可进行数据类型的检查。

四、在处理函数调用进行参数检查时，刚开始我运用一个全局的vector容器来存储函数的参数，在处理值参数表的时候，将每个参数加入该容器中。而后进行参数类型的检查，在检查结束后我会清空该容器，这就存在一个问题，若传入的参数中是一个函数调用，则会递归调用值参数表的处理，这样处理完该函数调用后，原来的容器会被清空，导致参数丢失。考虑到这个问题，我用了一个更加符合参数处理过程的全局栈结构stack<int> ParaType来进行参数的类型检查，但是要注意每次取出的参数的顺序，是从右向左的参数顺序，而且在完成参数检查后，仅将处理过的栈中参数pop，而不是将整个栈清空，这样便解决了上述参数丢失问题。

**3、中间代码生成阶段：**

中间代码生成较为简单，insertMiddleCode通过语法分析过程中的调用，实现将生成的四元式插入四元式表中，需要注意的是对于生成的临时变量，均以$开头，用于与代码中的变量进行区分。

**4、目标代码生成阶段：**

一、运行栈的设计问题，其中涉及到全局数据、局部数据的存储以及函数调用时现场的保存工作。我的运行栈结构如下图所示：

|  |
| --- |
| 上一级函数fp  函数返回地址ra  函数临时变量区  函数变量区  实参数区域（如果有实参） |
| 上一级函数fp  上一级函数返回地址ra  函数临时变量区  函数变量区  实参数区域（如果有实参） |
| `````` |
| 全局数据区 |
| `````` |
|  |

全局数据存放在$gp的相应区域，局部数据则从栈顶开始存储，发生函数调用时，先将当前函数的fp存储，然后将fp设置为当前的sp，然后存储函数的返回地址ra，之后进行调用函数的函数处理，在处理完函数调用后，将返回地址写回ra，再将sp赋值为当前的fp，最后将fp赋值为调用当前函数的函数的fp，最后jr $ra继续处理调用当前函数的函数。

二、变量表中的变量记录的地址是相对地址，由于分配栈空间是从高地址向低地址，故相对地址一般为负数，但是有一种情况例外，函数的形参的相对地址为正数，这是因为函数的形参实际对应的实参数据是在上一个函数压入栈中的，所以相对于当前函数而言需要从当前函数的帧指针fp向高地址查找，故偏移量为负数。

三、全局数据的相对基地址为gp，我在函数中运用一个临时寄存器$t9来代表$gp，故全局数据在获取时，其相对基地址为$t9；对于非全局数据，其索引地址即为当前函数帧指针，即$fp。

四、变量表存储的属性有name，kind，address，DataType，分别表示数据的名字，类型，相对地址以及数据类别，通过查询该表，获取到变量的相对地址，才能获取对应的数据。

**5、代码优化阶段：**

一、进行基本块的划分，划分方法参照书上的方法（三条规则确定入口语句），以函数为单位，进行基本块的划分，将基本块的划分结果存入basicBlockVector容器中。

二、DAG图处理时，对于每个节点，采用map<int, DagItem\*> dagIndexMap容器来建立索引，通过节点编号即可找到树中对应的节点，免除了树的遍历工作。

三、一开始，我采用结构体的形式来处理每一个节点，可最后发现利用结构体修改某个节点时，容器中的节点并不会被修改，且在每次循环中申请的结构体地址是同一地址，会出现覆盖的情况，故之后改为使用指针来处理节点，使用new DagItem()方法来申请新的节点，而后转为对地址的一系列操作即可。

四、在构造DAG图时，对于以变量为叶结点的结点来说，（例如a），需要将变量名字命名为a$，且节点表中加入a$，这是为了防止之后涉及到的a的修改数值操作而使计算出现问题的情况。

五、依据DAG图导出四元式时，对于临时变量：首先要遍历当前处理的语句所在基本块的后续语句用到的临时变量，对于后续需要用到的临时变量加入tempVarVector容器中。对于只包含临时变量的节点，需要保存一个临时变量作为该节点导出的符号；对于上述tempVarVector中存在的临时变量，需要保存这些临时变量；而其余的临时变量可以直接删除，不必导出。对于代码中的变量：若为上述添加的形如a$的变量，在节点表中查找a是否存在，若存在，说明a后续被修改了数值，则保存a$，添加a$ = a的赋值语句；否则，说明a后续没有被修改数值，则将节点表中的a$换回a，同时更新DAG图中的a$节点对应的信息。对于其余父节点的变量，则按照生成的导出序号队列逆序输出即可。最后，对于所有的叶节点，若当前节点还保存着除了节点名字以外的变量，则还需要设置该变量=叶节点对应的名字的变量的赋值语句。

六、寄存器分配采用引用计数法，以函数为单位，利用上述划分的基本块，对于出现的局部变量进行计数统计，而后进行排序，将出现次数最高的前8个变量分配寄存器（$s0-$s7），（如不足8个，则全部分配寄存器），而后将分配的变量寄存器映射关系存入FunctionRegMap中，之后在进行局部变量存取时，若该变量被分配了寄存器，则直接操作该寄存器即可，减少内存的访问次数。

七、常数合并是指将+ - \* / 指令的var1 var2两个操作数均为常数的指令进行运算得到运算结果，而后合并常数转换为赋值语句。例如加法：+, r1, r2, r3， 若 r1，r2为常数，则合并为 = , (r1+r2), , r3

八、冗余代码的删除实现方法为，将相邻的genLabel生成标签语句，除了最后一个语句生成标签外，其余生成语句均删除，并且将跳转到被删除标签的指令的part\_3标签替换为最后一个标签。

**6、错误处理：**

类似于语法分析的递归下降处理方法，错误处理依据程序的调用关系，在发生错误时，使当前调用的处理方法传入对应的错误类型，而后该方法返回，依据程序调用的上下关系，直到读取到满足下一条件的符号位置为止，继续进行处理。

不同错误的处理方案如下表所示：

|  |  |  |
| --- | --- | --- |
| 处理方案 | 对应的标识号 | 处理方法 |
| Quit | 0 | 程序退出，结束 |
| DoNothing | 1 | 不做处理，继续分析 |
| IntCharVoid | 2 | 读取int/char/void |
| ConstIntChar | 3 | 读取到const/int/char |
| CommaSemicolon | 4 | 读取到逗号或者分号 |
| LSmallBrack | 5 | 读取到左小括号 |
| RBigBrack | 6 | 读取到右大括号 |
| IfForLBigBrackICVISPSRS | 7 | 读取到if/for/int/char/void/return/scanf/printf/switch/左大括号/分号/标识符 |
| PlusMinusTimesDivSemicolon | 8 | 读取到加减乘除或者分号 |
| SemiColon | 9 | 读取到分号 |
| RSmallBrack | 10 | 读取到右小括号 |
| CaseDefaultRBigBrack | 11 | 读取到case或者default或者右大括号 |

大致的实验流程以及重点、难点如上述，在处理好上述各种的问题后，一个C0文法的编译器就大致完成了。