

《编译技术》 课程设计

申优文档

学号： ____17373423____

姓名： _____王少布_____

2019 年 12 月 16 日

目录：

目录：

一、功能架构设计

1.1 前期规划

1.2 学习准备

二、具体实现过程以及遇到的困难解决方案

2.1 词法分析部分

2.2 语法分析部分

2.3 错误处理

2.4 中间代码生成

2.5 目标代码生成

三、优化

3.1 全局寄存器分配（染色算法）

3.2 代码内联

3.3 常量传播和复制传播

3.4 无关变量消除（DCE：Dead Variables）

3.5 不可达代码消除（DCE：Unreachable Code）

3.6 窥孔优化

3.7 指令选择

四、实验感想

五、实现过程中参考过的文献

一、功能架构设计

1.1 前期规划

编译理论课伊始便为我们介绍了一个完整的编译器的组成。不同部分之间既有前后承接关系，通过一遍遍的处理来将源文件最终转换成可执行的汇编代码；又有一定的功能上的耦合：如贯穿大部分流程的符号表，和贯穿词法语法语义分析部分的错误处理程序。再加上考虑到设计前面流程时可能没有对后面内容充足的理论知识来支撑自己在最开始的时候进行完善的

设计。“解耦”必然成为重中之重。进一步地，“高内聚低耦合”和“为后续需求留足空间”成为了初期设计的优先考虑。

1.2 学习准备

理论指导实践，如果只是跟随理论课一步一个脚印的学习，虽然对于“当前”需求的理论是足够的，但始终还是会在面向“整体的”和“未来的”设计上欠考虑，因此第一步我认为我需要对整个流程有一个大致的概念，避免前面设计反倒成为后续设计的绊脚石。我因此首先在开始一切想法前将“龙书”《编译原理》1-8章通读，大致了解了编译器的工作原理和每一部分之间的相互关系。

二、具体实现过程以及遇到的困难解决方案

2.1 词法分析部分

词法部分是一切的开始，“龙书”以及我们的课件中提供了两种方式来处理这个单元：其一是将词法分析单独做一个流程，先进行一遍词法分析再进行一遍语法分析；其二是词法和语法合并为一个单元处理，词法分析向语法分析提供接口。两种方案各有利弊，我选择的是第一种方案，因为可以将词法和语法分析解耦，还适合增量开发，将词法语法分离也方便

Debug。

词法部分遇到的最大的问题是选择编程语言并熟悉语言新特性以及编程设计风格，由于经历过面向对象设计与构造的训练，我毫不犹豫地选择了C++来进行设计，设计风格必然是OOP。事实证明我的选择是正确的，C++提供的STL库使我免于重复造轮子，数据流与虚函数的语言特性也在我的实现中发挥着重要的作用。

2.2 语法分析部分

语法部分是第一个难关，我使用的是“递归下降子程序”法来对程序进行LL分析，递归下降方法虽然方便开发但是也同时带来了较大的开发量，需要有清晰的设计，对文法有相当的熟悉，以及可能的回溯处理。

我在语法分析部分遇到有两个棘手的问题：首先是关于设计，我在作业放出之前已经设计了一个基于规约与移进的LR分析器设计，因此在知道需要使用递归子程序法进行设计的时候需要面临重新设计的问题，还好之前对词法分析部分的处理具有相当的冗余可兼备支撑LL或LR分析器的设计，使我免于重构之苦。

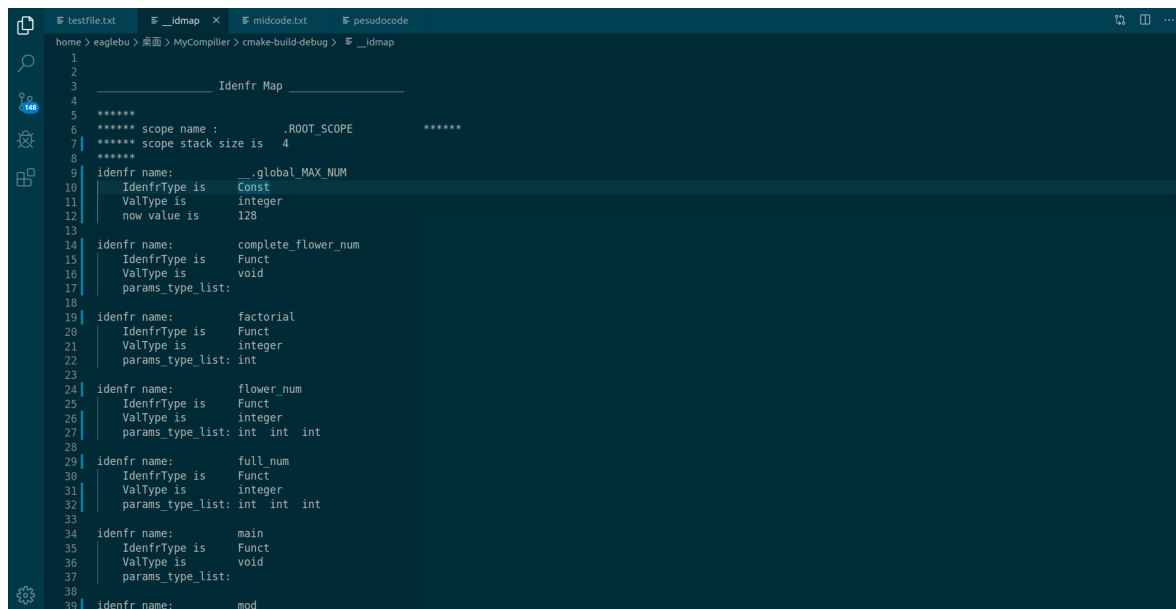
另一个问题是我该给下一步（中间代码）设计准备什么数据结构，虽然说对于属性翻译文法而言，“翻译动作”完全可以接在“分析”后进行，并不需要额外分家并且传递数据结构。但我考虑到递归下降程序法进行分析和进一步地形成中间代码的过程都整合了大量的代码，放在一起必然容易使得架构摇摇欲坠。所谓“不忘初心，方得始终”，我坚持使用了AST（abstract syntax tree）来使这两个过程解耦。虽然这开始备受争议，许多人指摘我说这么做没必要，但在错误处理（判断表达式类型）和中间代码生成时AST发挥了巨大的作用，许多耦合设计不敢进行的操作在我设计中也只需要轻松完成，最后我用事实让这些指摘不攻自破。

2.3 错误处理

有关错误处理，我和课程组的想法基本是一致的：一个完整的编译程序必然需要错误处理程序。但同时对于错误处理的规范和统一实在困难，因为正确的程序只有一种，而错误千奇百怪，要考虑到全部情况基本上是不可能的，要完完全全和课程组的想法保持一致更是困难，虽然课程组已经进行了“每行首个错误报对即可”的简化，但是如何使程序在出错后保持稳定继续运行到报对报准下一个错误依旧困难。

我在这个阶段设计和实现了符号表，我认为符号表对“名子”和相应的信息进行统一管理。并且在这个阶段将需要的信息归纳添加进符号表中。

符号表示意（Figure 1）：



```
1  
2  
3  
4  
5  
6 ***** scope name : .ROOT_SCOPE *****  
7 ***** scope stack size is 4 *****  
8  
9 idenfr name: _global_MAX_NUM  
10 | idenfrType is Const  
11 | ValType is integer  
12 | now value is 128  
13  
14 idenfr name: complete_flower_num  
15 | idenfrType is Funct  
16 | ValType is void  
17 | params_type_list:  
18  
19 idenfr name: factorial  
20 | idenfrType is Funct  
21 | ValType is integer  
22 | params_type_list: int  
23  
24 idenfr name: flower_num  
25 | idenfrType is Funct  
26 | ValType is integer  
27 | params_type_list: int int int  
28  
29 idenfr name: full_num  
30 | idenfrType is Funct  
31 | ValType is integer  
32 | params_type_list: int int int  
33  
34 idenfr name: main  
35 | idenfrType is Funct  
36 | ValType is void  
37 | params_type_list:  
38  
39 idenfr name: mod
```

Figure 1: 符号表示意，名字及对应的信息

2.4 中间代码生成

我对中间代码的定位是针对抽象语法树的一次针对化和扁平化处理，针对化指的是只针对抽象语法树中可执行单元（语句），其他的不会进入mips汇编程序.text段的信息要么功能结束被遗弃（如各种左右括号），要么已经被统筹进符号表中进行管理。扁平化指的是条件与循环会使程序运行流呈现树的形式，而我们需要将其转换成“线性顺序”才能装填进内存，因此需要对抽象语法树进行扁平化处理。并且考虑到我们的文法是类C文法，执行程序由函数组织得来，我的两级中间代码都有基于这一点的考虑，即代码由函数组成，函数再由语句组成，这个设计给我的目标代码生成（尤其是寄存器分配这一步）以及部分优化（函数内联）带来了很大方便。进一步的，我设计了两层中间代码，一层更贴近源程序的逻辑，即第一层一个中间代码语句代表了源程序中一步“功能”，为的是方便翻译流程；而第二层则更面向机器体系结构和统一化建模，即一个第二层中间代码对应了一条体系结构指令，而统一化建模（九元式）也使这种中间代码更便于进行一些优化与其他操作。

我遇到的第一个问题在于第一层中间代码的统一建模，我将大致语句功能分位8类，但是我需要将他们实现统一接口。我在这里使用了工厂模式，这极大的简化了我的翻译操作。同时，我也遇到了有关堆栈区的问题，概括地说就是在函数调用时创建的变量会保存在运行栈并在函数执行完毕后被回收，这产生了对象数据的丢失，我最后用指针和创建在堆上的内存解决了这个问题。第二个问题在与九元式的设计与第二层中间代码与体系结构代码区别和联系，我将这二者视作一个数据结构的两种不同表现方式，并

且将汇编代码视作对第二层中间代码的完善和补充，这样的处理方法虽然会在一定程度上使得第二层中间代码和汇编代码混淆，好在汇编代码就是程序的终点，不用再考虑后续问题。这同时为中间代码带来了灵活性，并且在实现中方便处理。

两层中间代码示意（Figure 2）：

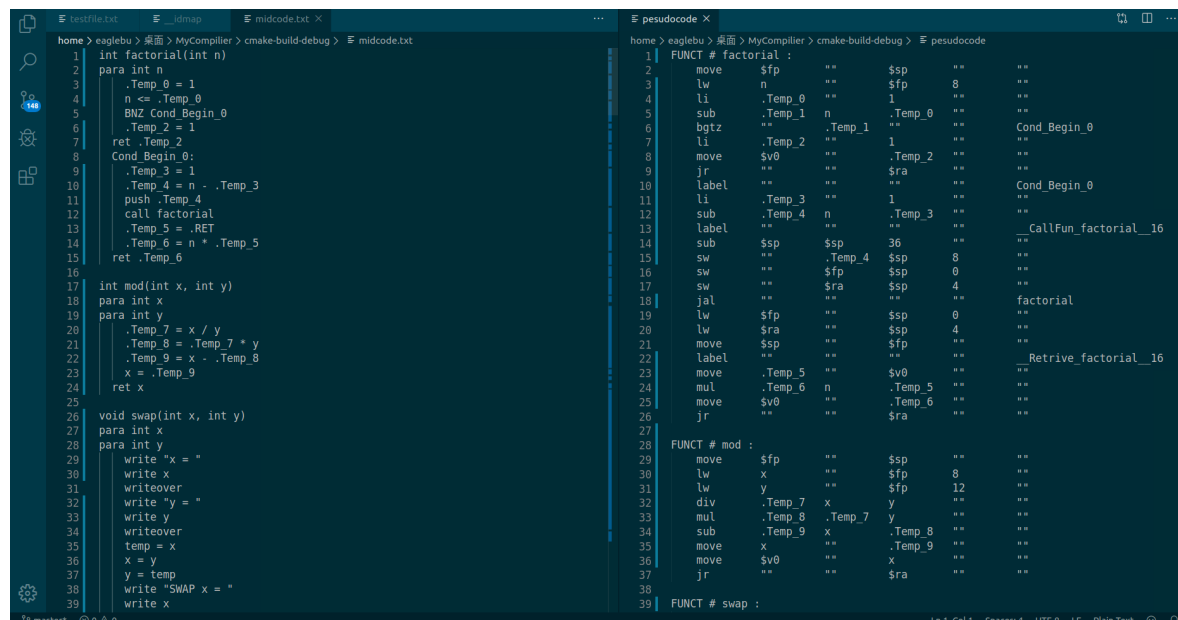


Figure 2：两层中间代码（左：官方Format 右：九元式Format）

2.5 目标代码生成

这个阶段建立在第二层中间代码之上，处理起来简单且舒服。最难的地方我认为在于良好的寄存器分配策略，这点我将放在优化部分详细谈。

遇到了一些问题比如mars支持的mips指令不支持rs区为立即数，需要在一定程度上对指令进行改变和扩展。

三、优化

3.1 全局寄存器分配（染色算法）

对于图染色法的全局寄存器分配，首先需要进行流图的制作，活跃变量分析以及到达定义分析，考虑到其他优化也会用到分析的结果，因此最好将结果用数据结构封装起来。我使用Dfs将后两者结合起来组织，方便实现，也为其他优化提供了方便。

对于图染色算法，课本上和课件上提供了Chaitin的染色算法，但考虑到这种染色算法具有一定的局限性，并且比较偏向悲观假设，效果可能不会太好。我查阅文献后选择了Briggs的改进染色法，这种染色法利用懒惰删除标记，将Chaitin染色法认为不能染色的节点依旧加入染色栈并且尝试染色，如果确实不能染色再确实删除，这种积极的算法解决了 Chaitin染色算法 中对于四节点环不能二染色的问题（Figure 3）。

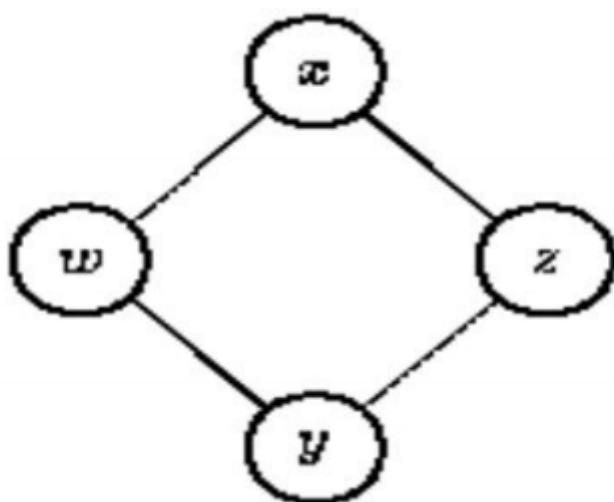


Figure 3 : Chaitin算法中四节点环不能二染色问题

3.2 代码内联

函数的相互调用是一个带环的有向树，代码内联中最大的难题也在于如何将这颗带有递归的函数调用树完全“展开”：我的解决方案基于一点考虑：即如果一个函数调用了其他函数，那么其他函数一定在这个函数之前声明并定义（若仍有疑惑请见“讨论区”我个人对函数调用提出的问题帖）。将当前函数完全内联（除了递归调用）的充分条件是他在他前面的这些函数定义都被完全内联。这就将这个问题转换成了一系列子问题，并且证明了只要按照函数声明顺序将函数逐个内联展开，那么我们就能实现完全内联。

进一步，与其考虑递归函数完全不内联（因为永远无法完全内联），又考虑递归层数是 $[1, +\infty)$ 即对递归函数进行一次内联必然不会冗余内联，因此我们与其判断什么时候什么函数是递归函数不如判断只要不是调用自身便可以根据上文算法进行内联。

这样便实现了完全内联展开并且没有冗余内联。

3.3 常量传播和复制传播

我的常量传播和赋值传播分为两步：

第一步是基本块视角，在基本块内部将常数和赋值操作传播，这个主要是基于对变量的定义使用链的分析（Figure 4），很多变量只被赋值了一次并且这个赋值（常量、复制）的使用都在基本块内，将这些赋值传播能使得相当比例的定义无用化。

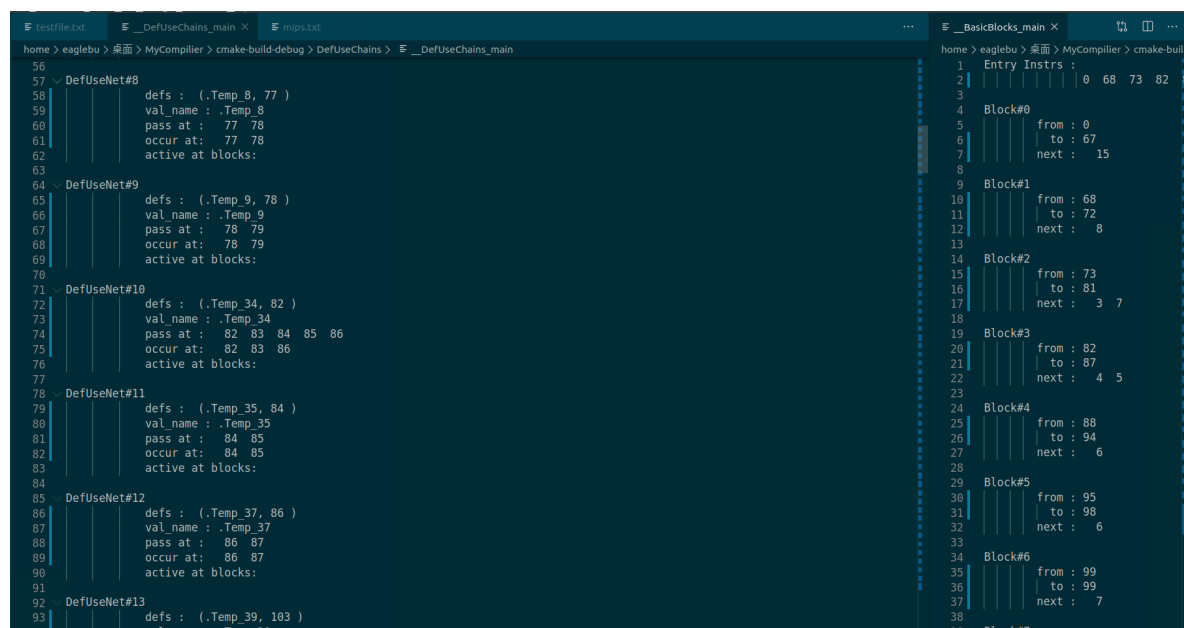


Figure 4: 未优化前的定义使用链情况（一共147个定义使用“网络”）

这步优化过后很多变量（尤其是临时变量）只剩下定义而没有使用（Figure 5），即只出现了一次。

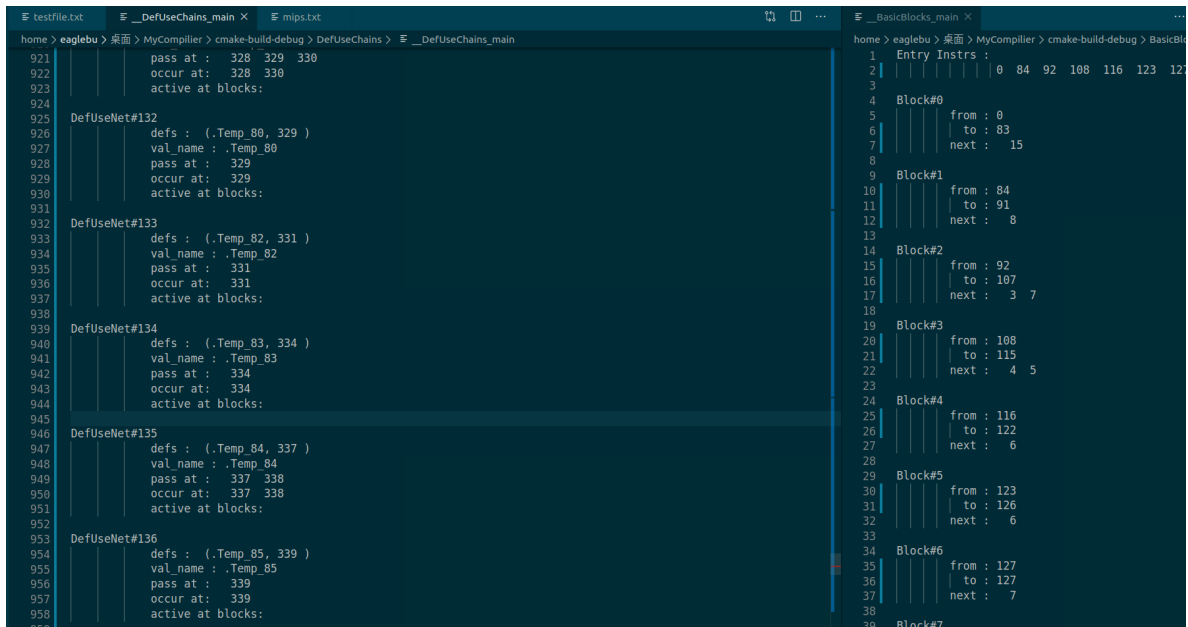


Figure 5: Spread化简后情况

然后在DCE一步，我们可以根据定义使用链将这些无关变量（Dead Variables）消除。

第二步是定义使用链（网）视角，首先先制造定义使用链和定义使用网，并且对每个只有一个定义的定义使用网，如果他符合传播格式（常量、赋值），就将其传播到每一个他的使用位置。这个属于第一步的深层次优化。

3.4 无关变量消除（DCE: Dead Variables）

常量传播和复制传播优化的作用只有配合着无关变量消除才能真正意义地体现出来。无关变量消除的条件很简单，如果这个变量是临时变量或局部变量，并且只有定义没有使用我们就将这个定义消除，效果如图（Figure 6），原来147个定义使用网络被减少到了只剩下59个。

The image shows two side-by-side terminal windows from a compiler's debug output. The left window, titled 'DefUseChains', displays a list of definitions and uses for various temporary variables (Temp_84, Temp_88, Temp_90, Temp_92, Temp_96) across different code blocks. It includes details like 'pass at', 'occur at', and 'active at blocks'. The right window, titled 'BasicBlocks', shows a control flow graph with 'Entry Instrs' and several blocks (Block#0 to Block#7). Each block contains instructions with their 'from' and 'to' block numbers, and a 'next' pointer. The output is formatted in a structured, tabular-like manner.

Figure 6 : 无关变量消除效果

3.5 不可达代码消除（DCE: Unreachable Code）

不可达代码有两个技术要点，一个是branch型指令的永真或用假判断，另一个是判断一个基本块是否是可达的。第一个算法比较多样，我是对branch条件是由常数或\$zero寄存器组成时进行判断。第二步因为考虑到未来可能的拓展性我使用了Floyd作判断，如果一个基本块可以直接到达另一个基本块，那么这两个基本块间有向距离为0，否则为1，最后和第一个基本块“距离”为0的都是可达的代码块。

3.6 窥孔优化

窥孔优化在我看来更多的是一些锦上添花的东西，和之前的优化相比技术含量小了很多，我认为也可以理解成一些小trick的集合。

我只添加了一个窥孔优化：

（1）如果这条语句是j型指令并且对应的跳转地址就是其下一条语句，那么将其抛弃

3.7 指令选择

指令选择更多的是一些锦上添花的东西，和之前的优化比技术含量小了很多，也可以理解成一些小trick的集合。原理如下：Mars对一些指令理解为扩展指令，这个功能很便捷但很多时候会带来性能的下降。我们可以自己进行这个过程。

比如将bge等branch指令换成一条计算指令和对计算结果的bgez指令，这样让计算指令进入优化作用域内会有一定的效果。

再比如Mars没有subi指令，但是有addi指令，我们可以将subi换成相应立即数相反数的addi，效果相同但是少一条指令。

最后是Mars自带的三地址div指令会自动扩展除零检查的相关指令，我们可以将这条指令自己扩展成没有除零检查的形式，因为这个div在循环里，所以除零检查多的一条指令被重复执行很多次，造成性能的下降。

四、实验感想

几乎陪伴我整个大三上学期的编译课设终于也接近了尾声，总结反思压力是有的，探索和求知是有的，随着一步步的增量开发，最终实现了一个能有应用价值的程序，喜悦必然也是有的。

五、实现过程中参考过的文献

- (1) Preston Briggs, Register Allocation via Graph Coloring, April 1992;
- (2) Steven S. Muchnick, Advanced compiler design implementation;
- (3) 武汉唐, 图染色法的寄存器分配