《编译技术》课程设计

申 优 文 档

学号：____16231086_____

姓名：_____白心宇_____

2019 年 1 月 1 日

目录

一、雏形构建

1. 千里之行，始于足下

编译原理课程设计是开始相对较晚的一门实验类课程。这其中势必有很多的考虑。编译系统作为一个庞大复杂的工程，需要前期坚实的课内知识支持才能够从一开始事半功倍。在编译原理理论课的一开始，我就体会到了这门课内容的复杂性。想要从头到尾的完整设计一个编译系统，需要结合高级程序代码、中间代码和目标平台特性等等因素才能够达到高水平的标准。虽然自己对 C 语言并不陌生，但是在一开始还是对整个系统的框架十分打怵的。

同时让我十分苦恼的还有模块化实现的分离性。每完成一个阶段性部分后，都难以验证这个已经完善的模块的正确性。除此之外先前实现好的模块还有可能因为之后的功能修改而需要重新改变架构，这对正确性又施加了新的考验。如何保证每个模块的正确性，成为我在每一步完成时首先需要考虑的问题。

在飞速飘过的九周之后，我们迎来了编译器课程设计的阶段性任务。每一行代码都是在每一天的努力中逐渐构筑起的，每一行都经过自己的反复斟酌。

2. 构建蓝图

在开始设计前，我先对我想要实现的内容进行了系统的思考。词法分析程序作为其他程序的真正输入，其内部实现不用过多考虑。但是词法分析程序作为源程序的直接接口应该具有检查错误并汇报的能力，因而在词法分析程序开始之前应规范好错误处理模块的统一化接口。为此，我在实现词法分析阶段性任务之前，便已经实现好了报错的格式和流程。对于错误的发生，我个人很喜欢 GCC 控制台输出的标明位置和出错代码的错误显示方式。因而在错误处理类中我实现了出错位置和错误类型的初步判断。

设计好错误汇报模块之后，我终于可以开始完成我的词法分析程序了。按书本中的描述按图索骥，根据抽取到的文法的特性提取关键字，生成关键字的查找数组并且定义不同符号的类型。因为之前已经实现好的错误处理接口，我可以在不正确的词法成分出及时汇报错误并根据情况跳读继续分析。在完成词法分析的过程中，我还是因为大意留下了隐患。由于被 C 语言洗脑过度，我本能的以为字符串中允许的字符和字符型变量中允许的范围相同，导致我没有在词法分析阶段及时报错。这个问题直到语法分析阶段才被我发现，算是疏漏的一点。

完成词法分析之后，就要开始语法和语义分析部分了。这两部分在实现前，考虑到常量合并和常量传播和语义的关系实在是太接近，因而实现中就将对常数的处理预先内置以减少之后的返工。同时，语义分析过程已经开始和中间代码的生成紧密相连，故开始工作之前我先对中间代码四元式的设计进行了规范。在最初设计四元式时，简单的仅设计了 BEQ 和 BNE 类判断的跳转。之后再目标代码生成时，发现四元式的局限性导致了目标代码的局限性，使我必须用很多运算类指令来弥补，因而对四元式序列进行了修改以支持更多类型的指令转换。

最后的目标代码生成也是我反复返工斟酌的部分。最初的实现很简单的把赋值语句分为三部分，即目标，左运算符和右运算符。这么做无形中出现了很多不必要的操作。包括函数传参过程中，我也使用了很多冗余的指令，对栈指针的操作也为了方便有很多的重复。这些在后期优化时都被我一一更正，自己也为自己当初草率的编码付出了时间的代价。

二、精进优化

1. 细节处见成败

第一次编译成功并运行代码，我的代码量是我同学的两倍，运行周期数则是四倍。

内心不服的我拿来了同学生成的目标代码，和我的目标代码逐行比对，发现自己在草率实现的过程中偷过的懒全都找回了自己身上。首先最明显的就是函数的压栈处理。粗暴的将寄存器全部压栈的操作显然是十分不可取的，有选择的才是明智的。其次是对体系结构的利用。之前的代码中几乎使用了静态的地址分配，在对内存操作时竟然忽略了寄存器-寄存器式体系架构的优越之处，没有使用全局指针、栈指针等内置的指针，反而单纯用立即数操作内存，在 MARS 执行期被翻译成了三条甚至更多的指令。还有就是对常数的处理。MARS 允许运算类指令和立即数混合运算的操作，而我的代码却出于简单考虑统统赋予寄存器，导致前期常量合并的优化功亏一篑。

在更正了代码中过分的偷懒导致的惨剧后，我又开始思考一些更精进的优化方式。首先是对函数帧指针的优化。这个变量在每一次函数调用时都需要压栈，取出之后还需要进行计算。但是对于静态程序语言而言，活动记录的大小在扫描目标代码时就已经确认不会再更改，因而其实函数栈指针的数值是可以确定的。激动的我去掉了函数调用时的两条压栈指令，总周期数又下降了一些。

在研究 MARS 运行代码的行为时，无意中发现除法指令另有玄机。原来 MARS 为了防止出现除 0 错误，每一个和寄存器相关的除法指令都会展开出一条分支指令。要知道分支指令的权重是 2，这样一来就严重降低了代码运行的效率。将除法指令拆分成三元除法指令和从低位取数字即可避免这个问题。

函数压栈还有后续。我觉得虽然我一改之前什么都压栈的习惯，但是还可以更个性化定制一点。一个寄存器什么时候需要压栈呢？应该是当且仅当这个寄存器在本函数层中被使用且这个寄存器在被调用者的函数中使用才会被压栈。这样很多临时寄存器的压栈都可以被优化掉。

之前的实现中，采取将函数参数压入内存中的传递方法。但是考虑到 MIPS 体系结构为我们准备好的参数寄存器，决定将采用参数寄存器作为媒介传参的方式，又减少了无用的内存访问。进入函数之后，若该变量在函数内部被分配了寄存器则直接移动到目标即可，可以不经过内存。

2. 正统的优化

对教材的深入研究之后，决定实现教材中的 DAG 图与寄存器分配的优化方法。之所以选取这两种方法，是因为我觉得计算作为程序的主要功能，一定会频繁用到。而如果可以固定减少某些部分的运算，那么成绩也一定可以降低。寄存器分配则是 MIPS 体系结构最重要的一环。合理的寄存器分配可以显著的降低对内存的访问，提高程序运行速度。

2.1 DAG 图

在具体的实现中发现 DAG 图真的有很多潜在容易犯错的问题。最后实现的 DAG 图版本也是在一次又一次测试中完善的。

2.2 引用计数

在全局流图和引用计数方法之间徘徊了许久不知道如何是好，最终选择了引用计数。选择的原因有二。一是因为引用计数法的运行效率上速度更快，且不会被复杂的循环等拖慢速度。第二是引用计数法比较好继承之后 Profiling 部分的实现，达到更高的效率。

3. 旁门左道

实现了寄存器分配算法之后，我发现还是有很多难以覆盖的特殊情况。受书中引用计数法分配寄存器的启发，实现了运行时优化部分和小函数调用时的内联机制。

3.1 运行时优化

经过广泛的查找资料以及和老师的沟通，确定了运行时优化是可以采取的一种方式。编写代码的初衷是因为 MARS 的指令统计速度实在是太慢了，增加了我每一次检验成果的时间。之后发现模拟器的完备性已经可以支持我将其纳入到我的编译器中，作为编译的一部分，所以才萌生了运行时优化(Profiling)的想法。

程序在模拟执行时会记录变量的访问信息与调用堆栈信息。这样对于在递归调用的函数中使用的寄存器，程序可以判断这个变量是否有被分配寄存器的价值，以及它在寄存器分配队列中的优先级。高优先级的变量会更容易得到寄存器的分配，因而提高程序运行效率的效果更显著。

3.2 函数内联

观察目标代码运行时，发现对于小函数或功能型函数而言，频繁调用过程中的压栈开销不容忽视。因而选择将小函数内联至目标代码中的想法。

对这些函数，首先总结一下他们的共同特点。它们内部的变量个数较少且不涉及数组型变量，函数内部不会再调用其他函数。因而针对这一类函数，我选择使用内联优化，减少频繁调用过程中产生的反复现场保存以及参数的转移。为了减少内联函数寄存器转移的开销，内联后的部分中涉及的所有变量都按符号表中的顺序分配函数参数寄存器（对于不是参数的变量同样可以分配参数寄存器使用）。这样实现的内联操作允许内联函数中存在四个及以下的变量。在内联优化的实现过程中遇到并解决了以下的问题。

局部变量映射问题

内联函数内部的变量需要映射到参数寄存器上并正确的参与运算。

临时寄存器覆盖问题

进入内联函数体前，需要将外层函数和内联函数都使用的临时变量寄存器保存，在跳出内联函数后恢复。同样，对于需要使用参数寄存器的输出语句，需要将参数寄存器暂存，在输出语句结束后恢复参数寄存器的值。

内联标签冲突问题

对于内联函数中的标签，为了避免同一个函数在多处被内联后造成标签重名问题，在进行内联替换时将标签后追加计数用来区分不同内联部分的跳转标签。为了防止内联标签和某些用户定义的函数标签重名，实现时将内联函数标签头部追加 $ 符号用以区分。

内联函数多出口问题

为了解决内联函数的多出口问题，优化实现时对每一个内联函数的返回语句追加一条跳转至当前内联块结尾的跳转语句。同样，跳转的目标标签也要解决标签冲突问题。除了这个

问题之外，还要考虑内联函数出口标签和内联函数名区分的问题。实现上通过添加前缀 $_ ，辅以上个问题中标签结尾追加的引用计数即可唯一确定一处内联的出口。

在解决了以上问题之后，功能型小函数得以成功内联并减少了频繁的压栈次数，使效率更上一层楼。

三、实验感想

编译原理课程设计是北航计算机专业三大核心中鼎立门户的专业课。继承了计算机组成设计的 MIPS 体系架构和操作系统课程中的 JOS 小操作系统对硬件的控制逻辑，使得编译原理课程设计可以站在硬件之上，做针对体系结构的拓展和优化。

作为一个核心硬课，课内所教授的知识点更偏向应用实践，可能是由于编译本身是一门技术而言的吧。教材中的很多分析法、自动化方法都是在工程中得以应用的真实案例。对于我们而言，教材中的思维方法和设计模式就是我们效仿的模范。我们在理论课的指导下开始编写我们自己的编译器，从文法开始慢慢分析理解 C0 文法的限制和意义。词法分析、语法分析、语义分析和中间代码生成、代码优化和目标代码生成，这五个部分在课程设计中也一直在指导我们向前探索。

建立词法分析器是对教材自顶向下分析法（递归子程序法）的一个直接应用，经过自己的思考，我还借鉴了属性翻译文法对于计算类表达式的优化，通过将波兰中缀表达式转化为后缀表达式即可利用计算机的栈式结构直接计算正确的结果。对于前缀的修饰符号，我通过添加前导 0 的方法巧妙解决了这个特殊情况。这段程序正巧和理论课学习时我编写的练习程序不谋而合，因而正好嵌入到了编译器中。

在真实的工程中，我发现语法分析和语义分析之间很难划分出具体的界限，因而我将他们合二为一的进行。在语法分析的同时进行语义分析并生成中间代码。中间代码的设计在最初出现了极度冗余的情况，通过和同学们讨论，我发现我的很多中间代码都可以进行优化合并，从而推出了现在这套较为简洁的中间代码。在每一次的调试中，我发现中间代码是用来检验代码正确性的很重要的指标，正确的利用中间代码便可以及时准确的发现程序的错误所在。

通过中间代码生成目标代码的过程也是一个精心打磨的过程。最初的草率和鲁莽导致我在翻译的时候没有考虑体系结构的优势和特征，使得我翻译的每一段话其实都会成为 MARS 中的伪代码，翻译出两个甚至三个指令执行。最初的版本上线之后 8800 条总指令执行数目的结果和同学们一对比，发现竟然比他们足足多了 6000 余条。深受打击的我再次返工每一条中间代码的翻译，精心打磨，思考代码层次上针对体系结构上的优化。经过精简之后，代码总执行数目缩减为 4000 条，删除了近一半冗余和复杂代码。

在此基础之上，我准备开始着手进行进一步的优化。首先映入眼帘的就是常量预处理类的优化。由于之前良好的体系结构，我可以比较轻松的实现常量在词法分析中的预处理；除此之外，理论课学过的 DAG 图优化也让我记忆犹新。利用导出后的中间代码我可以启发式的化简和重新导出中间序列，在实战中非常实用；最后就是最重要的寄存器分配策略。如何利用好 MIPS 体系结构提供的寄存器，是能否提高程序运行速度的瓶颈。在寄存器利用方面，及时重置临时寄存器、巧妙利用全局寄存器以及选择性压栈、利用参数寄存器传参等操作都大幅度的降低访存数量，提高最后的评分。最后，利用内联函数的方法减少小函数调用时的压栈等行为，进一步优化指令的运行效率。经过不懈的努力，我终于将最终的总指令数目降到了 2400 条，实现了近 1/4 的优化成效。

编译原理课程设计是将已学知识巩固扎实并投入实践的最好机会，是编译课程深入计算机专业的窗口。通过编译课设的训练，我对编译技术的知识也有了更透彻的理解，虽然在完成的过程中遇到了很多约定不一致的问题，但是最终的实现还是非常有成就感的。希望编译的体系能够越来越完善，提供越来越好的学习体验。