**C0编译器**

在这个文档中，我将主要介绍如何完成一个C0文法的基于mips的编译器，以及在project进行过程中遇到的问题和解决方案。本文按照项目完成的流程来具体介绍。

1. **分析文法**

在开始设计之前，把文法研究透彻是十分重要的，这将大大影响后续工作的效率。

首先拿到文法之后需要从按照一定的规则概括分析，比如我是按照自顶向下的方法，从<程序>这个识别符号开始，逐步细化，一直分析到字符串、数字等基本元素。在有了大概的概念之后，就要开始对细节部分有所重视，比如字符串的组成不包含哪些字符，数字前的正负号、是否区分大小写等等。

**2、语法分析**

在对文法有了清楚的分析之后，就要开始设计语法分析程序（包括词法分析程序），

在之前对文法的分析就起到了作用，我们需要判断该文法是否有左递归、是否存在二义性等。对于有左递归的需要改写为非左递归。存在二义性的需要去除二义性。而在C0文法中，二义性还是有一些的，比如＜常量定义＞::= int＜标识符＞……与＜声明头部＞::= int＜标识符＞……这时就需要分析如何去除二义性了。最常见的有两种办法，向前预读字符或者直接采用BNF扩充文法来改写文法（提取相同的首部部分）。经过分析，我认为采取前者比较好，因为如果改写文法，将导致递归下降子程序的设计出现麻烦，每条文法的语义会变得不明确，随之还将给后续的语义分析带来很多麻烦。在这里可以采取使用一个预读buffer数组，存储所有预读的单词，在遇到二义性的地方，只需向前查看即可。

另外，在编写语法分析程序的过程中，最好就开始考虑错误处理，至少是语法错误。因为语法分析程序是整个编译器的框架，如果等做完了再开始设计错误处理，会对语法分析程序有所改动，比较麻烦。

**3、语义分析**

如果说语法分析是整个编译器的骨架，那语义分析就是编译器的血与肉，在项目中

是最重要的成分，由于语义分析涉及的东西很多，下面分开介绍。

1. **符号表**

符号表是编译程序分析代码过程中必不可少的帮手，它需要记录所有在程序中出现

的必要信息。由于该C0编译器要做优化，所以意味着需要中间代码，这也意味着需要“2遍扫描”。第一遍对原代码扫描，翻译得到中间代码，第二遍对中间代码扫描，翻译得到目标代码。在这两遍翻译过程中，都需要单词详细信息的支持，所以我在这里设计的符号表不做退栈处理，完整符号表一直保留到编译完成。使其不仅仅对第一次扫描有效，对第二次扫描也有一定的帮助。这样设计有好处，就是第二遍对中间代码扫描的时候，能获取最多的单词信息，也有坏处，即符号表占据的系统内存较多。

符号表不退栈是可行的，比较于pascal文法，需要的退栈是因为函数模块的嵌套导致的，由嵌套关系导致的需要对变量的可见性与有效性需要与嵌套关系一致。而C语言不允许函数的嵌套定义（不是嵌套调用），函数之间的关系是平等的。由此，一个函数中如果未找到变量a，只能去全局变量区寻找。基于这种关系，我将符号表设计为以函数为单元收集变量、常量等信息。全局变量也由一个特殊函数单元保存（该函数不存在，仅仅是为了与其它符号表保持一致），这样，寻找变量只需要在本函数和特殊函数中进行就可以了。

1. **运行栈**

运行栈是语义分析的核心。设计好的运行栈十分必要。本编译器的运行栈借鉴了

mips官方的运行栈结构，也做了一些简化。为了方便处理和理解，将官方规定的栈空间向下生长改为向上生长。在函数的传参方面，官方规定$a0到$a3为传参寄存器，多余参数通过内存传递，而在本编译器中，所有参数统一采用运行栈传递的方式。

在设计过程中，遇到了一个棘手的问题就是寄存器的保存和函数序言。在这个问题上耗费了很多时间，代码在这部分也出了很多bug，下面来介绍一下。所谓函数序言就是在函数刚进入的时候固定要做的工作，包括将保存调用者的栈指针、帧指针、返回地址、保留寄存器等。我在编程实现的时候，仅仅考虑到了栈帧指针，忘记保存函数返回地址，结果普通函数调用没有出现问题，但一旦有多个函数嵌套调用，就会导致外层函数找不到返回地址，导致程序崩溃。

1. **中间代码设计**

中间代码是介于源语言和目标语言之间的语言，它的存在主要是为了优化方便。我

采用的是四元式，方便翻译成mips汇编。在设计四元式的过程中，遇到的主要麻烦有两个。其一是分支语句如何设计，其二是函数返回。

由于mips里没有大于跳转小于跳转等指令，所以在设计分支语句的时候有一个trade off。选择偏向源语言还是偏向目标语言。前者的话，可以直接将分支表示为类似：GTR a,b,tag的形式，这样的话就需要在翻译为mips的时候多做工作，既比较又跳转。后者则将分支拆开成为两部分，比较和跳转，类似GTR a, b, c 和 BGEZ c, - , tag，这样的好处是在第二遍扫描时工作较少，不易出错。在我的编译器里采取了后者。

分析文法可以发现，函数的返回存在隐式返回，即void型的函数没有return的时候，也可以返回。所以为了在没有return的时候让编译器知道需要返回，设计一个函数结束标志是有必要的，我在四元式里设计了一个FOOT的式子，用来标识函数的结束。

四元式的操作数的数据结构也很重要，在此我将符号表中的符号项的数据结构同时用于四元式的操作数，四元式的部分操作数直接来自于符号表的元素，另有部分比如string、tag由四元式单独管理。这样有一个好处就是从语义分析到优化，都可以随时随地得到一个变量、常量的详细信息。

**4、优化**

优化是编译里面可选的部分，做优化可能导致很多意想不到的bug，而且随着优化

的进行，中间代码和目标代码都变得不再语义清晰，会给调试带来很多麻烦。但优化的好坏是编译器的性能优劣的体现，所以好的优化还是很有必要的。

1. **块内优化**

块内优化我主要做了常量合并和传播，关于常量的传播早在设计四元式的时候就做

到了。对于const常量，的我采取和字面量相同的管理方式，即直接在四元式里保留它们的值，这样在翻译的时候也就实现了常量的传播。而常量合并则需要在生成四元式的时候做检查，如果发现操作数都是常量，则由编译器进行运算，得出计算结果。

1. **全局优化**

分为活跃变量分析和全局寄存器分配，活跃变量分析采用了计算in集合和out集合

的方法完成。全局寄存器的分配则由图染色算法完成。这里需要注意的一点是，全局寄存器不由寄存器池管理，所以对他们的管理需要在mips代码中手工实现，比如在进入一个函数的时候统一将调用方的全局寄存器保存至内存，再载入自己的全局寄存器。再函数结束之前，需要将自己的全局寄存器写回，再将调用者的寄存器恢复（有点像操作系统里的上下文切换），另外，还需要注意将已分配全局寄存器的变量标识出来，免得再被分配临时寄存器。

1. **临时寄存器管理**

这一部分是最容易出bug的地方。临时寄存器只在块内有效，所以离开基本块之前

必须写回所有寄存器。在每个变量使用之前，都要申请临时寄存器。我在写这部分的代码时，遇到了一个很隐蔽的bug：同时申请2、3个寄存器时，存在后一个寄存器覆盖前一个寄存器的情况，为此，我在普通的寄存器池的数据结构上做了一些改变，增加了记录最近新使用的寄存器的数组，并规定最新使用的三个寄存器为保护寄存器，禁止分配，问题解决。

**5、错误处理**

由于在语法分析的时候，就搭建好了语法错误的框架，所以只需要实现即可（语义

错误类似）。这里主要说明出错之后的跳读。跳读的原则应该是尽可能少跳，将错误尽可能的控制在单句语句中，防止由一个小错误引起连环错误。

另外，为了防止错误信息过多重复，规定当错误数超过一定数目，将不再报错。这也和平时使用的编译器一致。

**6、调试与测试**

1. **测试**

测试样例的设计是减少bug的重要保障。我认为，应该根据语法和语义了两方面进

行设计，语法分析自不必说，按照文法进行覆盖，注意++2、-+1以及其它不容易注意的细节即可。语义测试，则有很多，每次只需针对单个功能，比如函数递归调用、函数传参、函数返回、表达式计算、全局变量与局部变量的引用、数组嵌套索引等等。

1. **调试**

在整个项目中，我主要调试的方法就是打印，包括符号表、四元式、活跃变量、基

本块、冲突图、寄存器池等，另外在汇编代码中输出注释可以很好的快速定位翻译汇编部分的错误。

**7、总结**

在完成了整个课程设计之后，我感觉在各个环节中，前期的设计十分重要，好的设

计可以保证后续工作都是连贯的，而非建了拆，拆了建。另外，适当的参考官方的设计思想也很有必要，比如运行栈的设计以及寄存器的使用规则，mips的官方就已经给出了很到的建议，我们要做的只是稍作改动，使之适合我们的环境，这样做也有一个好处，就是遵循统一的接口，比如大家都用$16~23做全局寄存器，我们却非不这样做，就会导致编译器的兼容性很差。