实习报告

吴耀轩, 1800012913

1 编译器概述

1.1 基本功能

编译器实现了从 sysY 到 eeyore/tigger/asm 的转化，并完成了阶段的整合。

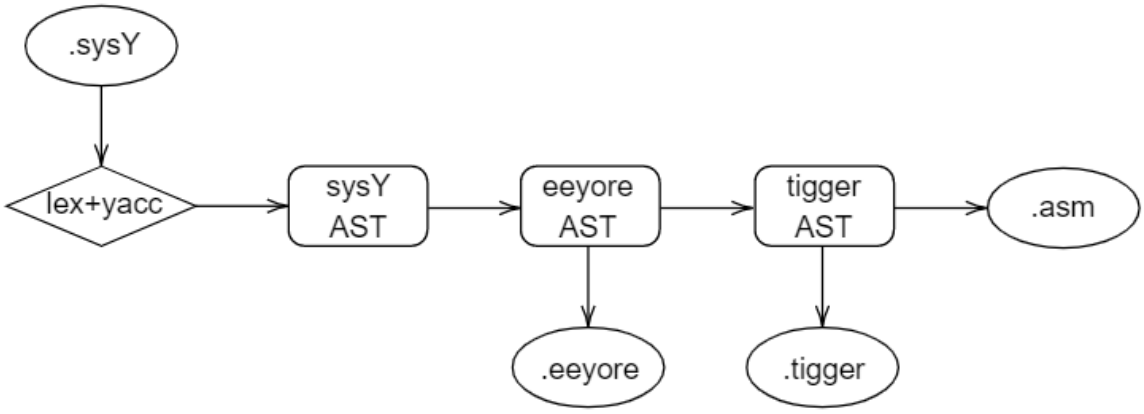
1.2 所开发的编译器特点

编译速度快。实现了常用的流敏感数据流分析和寄存器分配。针对性能测试的数据特点，作了一些 triky work。整体实现的框架比较清晰，后续添加优化比较方便。

2 编译器设计

2.1 概要设计

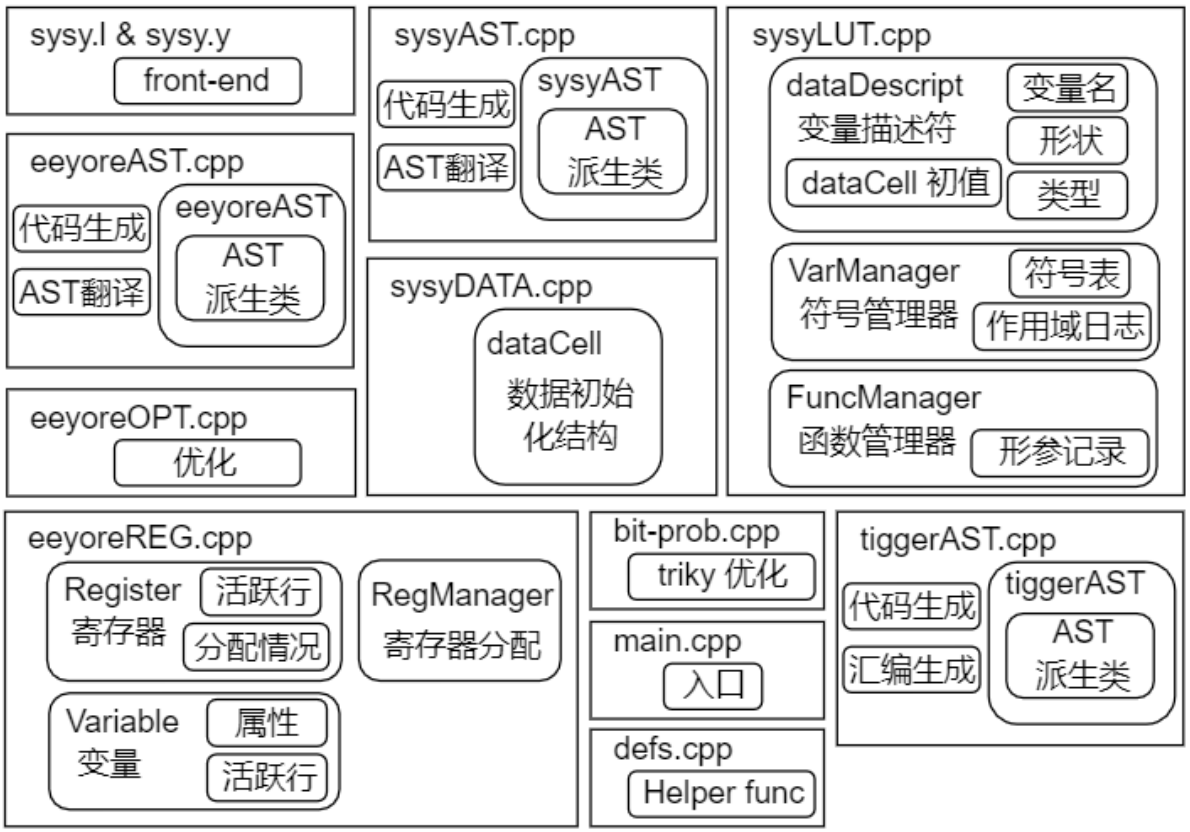
2.1.1 主要模块组成



编译器的实现主体划分为三个阶段。

- 前端：由 lex+yacc 将.sy 文本转化成 sysY 语言的 AST 形式。
- 通过翻译器，直接将 sysY 的 AST 转化为 eeyore 的 AST，主要完成转三地址码、多维数组压成一维、变量名重整和优化。
- 通过翻译器，直接将 eeyore 的 AST 转化为 tigger 的 AST，主要完成寄存器分配。
- 后端：直接在 tigger 的 AST 上 on-the-fly 生成出汇编文件.asm。

2.1.2 主要数据结构

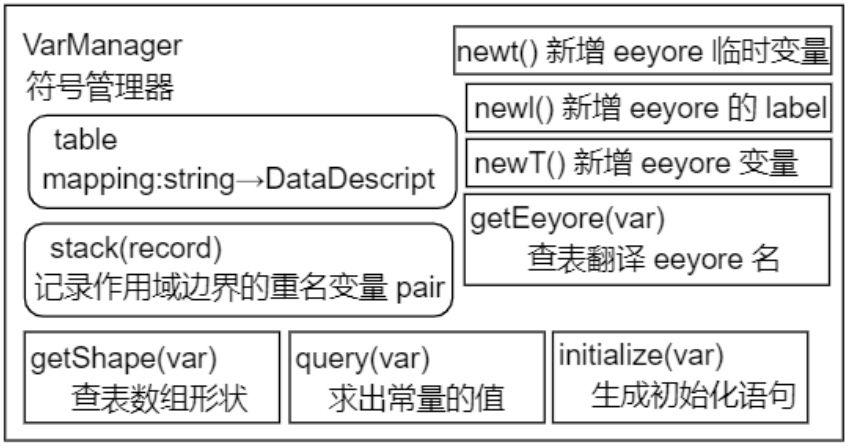


主要使用的数据结构包括。

- 三阶段的 AST：抽象语法树记录了对应节点的信息，外层是 sysyAST/eeyoreAST/tiggerAST 进行封装，针对不同的节点类型，有不同的派生类描述信息。AST 主要的方法有两个，打印代码函数 Dump() 和翻译到下一阶段 AST 的函数 translate()。
- 初始化信息 dataCell：多维数组是一个树形结构，在 lex+yacc 完成转换后，一个多维数组存放在 eeyoreAST 子类 _TREE 中，但此时是左孩子右兄弟的形式。为方便后续操作（初始化），将其转成封装好的 dataCell 形式。
- 变量描述符 DataDescript：描述符记录了变量的类型（全局/局部，数组/变量），数组的维度和初始化信息（dataCell 类型）。
- 符号管理器 VarManager：记录 sysY 程序中符号对应的变量信息（变量信息存放在变量描述符 DataDescript）。为解决重名变量，有日志文件记录进出作用域块时新增变量的信息，退出作用域块时，将消解的变量恢复到作用域块外最近的变量。同时，记录的变量属性为后续程序分析和寄存器分配提供支持。主要的方法有：初始化函数 initilize()，查询变量属性的函数以及支持分配 eeyore 中变量和 label 的操作。
- 函数管理器 FuncMananger：记录函数的形参个数和形参属性（变量/数组）。
- 寄存器管理器 RegManager：完成寄存器的分配，记录寄存器的分配情况。

2.2 详细设计

2.2.1 SysY 的符号表



本编译器将实体变量的信息，如全局/局部类型、数组形状、初始值都存储在变量描述符 DataDescript 中。在进行程序转换时，需要将变量名从 sysY 翻译到对应的 eeyore 名，此时需要通过变量名定位其对应的变量的描述符 DataDescript。

变量名到 DataDescript 的映射关系可以用 map 容器维护。但，外层作用域和内层的变量重名会导致映射出错。这里的解决方法是，在进出作用域块时，用一个日志记录发生了那些重名变量的替换，将变量名映射到内部变量的 DataDescript；在退出对应作用域块时，通过日志文件，恢复外层的映射关系。

2.2.2 寄存器分配

寄存器分配采用以下准则。

- 保留%t8,%t9,%t10,%t11 这四个寄存器以供临时存放变量之用。
- 全局变量都分配寄存器。
- 局部变量分配时不考虑过程间的关系。计算每个变量的活跃行，贪心地按照活跃行数从多至少，分配寄存器。每次尝试分配时，尽量配给到已分配的相容寄存器（活跃行不交）；实在不行，则分配到一个新寄存器。
- 若不能分配，则放到栈帧上。当需要使用时，利用预留的临时寄存器暂存。

特别的，变量的活跃行和优化部分一起实现，本节不作介绍。

2.2.3 Eeyore 的符号表

Eeyore 阶段的符号表，主要用来记录变量的类型信息，如是否为全局、常量，以便后续优化；另一方面，符号表需要记录数据流分析的信息，不过这部分的实现比 SysY 的符号表要简单很多。在实现时，eeyore 的符号表存放在模块 RegManager 中。

2.2.4 优化

本编译器的优化主要体现在四个部分：sysY 转换到 eeyore 阶段的优化、eeyore 层面的优化、针对数据优化、汇编语句的优化。最终性能测试在 78s 左右。

2.2.5 sysY 转换到 eeyore 阶段的优化

由于 eeyore 是三地址码形式，因此赋值语句在转换时要保证左值和右值至少有一个是单变量。比较简易的实现方法为，将赋值语句的右值先存放到临时变量中，再使用一条语句赋值。但其对赋值语句如 $a = b + c$ 的情况（当左值本身是单变量时，eeyore 允许右值可以是一次变量间运算、数组访问、函数调用），上述实现方法会多引入一个临时变量 t，先把 $b + c$ 传递给 t，再将 t 赋值给 a。如此做法会造成一次额外赋值，造成不必要的时间开销，且这里因为有数据依赖，会造成流水线停顿。

所以，sysY 阶段在转换到 eeyore 时，就应当减少额外的计算。

2.2.6 eeyore 层面的优化

之所以在 eeyore 阶段衔接主要的数据流分析，是因为三地址码形式整洁，可以避免讨论。为了简单，这里仅作了过程内的流敏感分析。首先扫描每个程序得到控制流图，在控制流图上求出通过 bfs 得到所有的 def-use 对，时间复杂度 $O(n^2)$。由于代码行数 n 比较小，所以控制流图的建立以及优化部分都比较粗暴，速度仍然极快。

获取 def-use 对后，可以作活跃变量分析、活性分析、常量传播，此时变量的活跃行也被求出，可以供寄存器分配模块使用。特别的，对于一个变量赋值为函数调用的情况，即使该变量此后无用，但函数本身可能有副作用，所以仍然要执行函数调用。进一步的，本编译器实现了简单的副作用分析，即当函数仅调用无副作用函数，无访问数组，无调用和读取非常值全局变量时，其也是无副作用的。

另外，编译器还做了赋值消减。如 $c = a + b, d = c$ 而 c 未被再使用时，可以直接改写成语句 $d = a + b$。

再者，编译器也实现了循环常量外提，循环展开（减少跳转语句）。

2.2.7 针对数据优化

针对性能测试数据的 FFT 部分，其瓶颈在于内部循环实现的快速乘。编译器可以通过简单的程序分析（实现上实际是针对测试用例的模式匹配），分析出函数摘要。具体地，FFT 内部的的快速乘 multiply() 可以分析出是按照比特进行的分治算法，故能改写成循环形式。

另外，编译器还实现了比较 triky 的优化：快速乘的中间结果都是正数，所以取模操作用 1 次 remu 代替 4 次的 rem 操作；针对手写生成的优化版本的快速乘汇编程序，还加上了乱序执行以减少流水线停顿。

2.2.8 汇编语句的优化

将 $*2^n, /2^n, \%2^n$ 等用位运算代替，消减运算强度；将 $+0, *0$ 用赋值语句替代，效果甚好。

2.2.9 类型检查

编译器在数组访问和求数组地址时，都会按照变量描述符里记录的数组形状判断是否访问是否合法。但实现时只进行了一系列 assert，未完善编译错误信息的输出。

3 编译器实现

3.1 所涉工具软件的介绍

3.1.1 功能

lex + yacc 套装。将无格式的文本转换成抽象语法树。

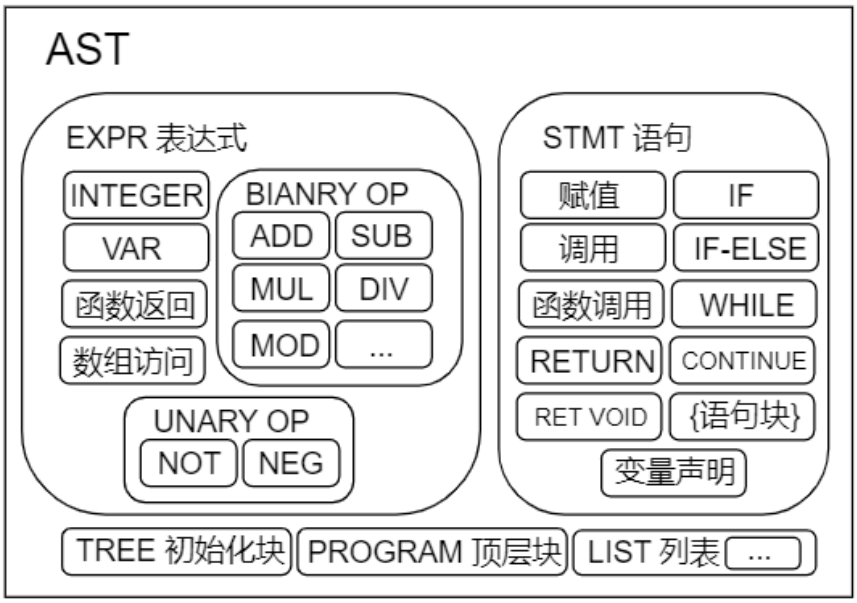
3.1.2 自己的看法

实际上，编译器开发者完全可以通过手写前端来处理文本。前端，本质上就是一些简单但麻烦的字符串处理，而 lex+yacc 的存在让编译器的实现可以绕过字符串处理，直接在比较简洁的形式上操作。

从我的角度来说，lex+yacc 就是一个格式产生器，每一条产生式就是一个格式。但 lex+yacc 也是有弊端的，该工具已经存在多年，其执行效率远比不上手写的前端；而 chord 等程序分析工具可以支持直接转换到比较高级的 IR，并支持插装，以进行更高级的程序分析和优化工作。尽管如此，lex+yacc 对实现一个简单的编译器来说，已经完全足够。

3.2 各个阶段的编码细节

3.2.1 抽象语法树的实现



外层分别用 sysyAST/eeyoreAST/tiggerAST 封装一个类。而内层有多个派生类，并用多态调用的方式解决不同类型抽象语法树节点的操作问题，可以简化一些代码实现。（有一些节点类型可以复用一个实现，但多态也有弊端，每次加一个新方法都要引入一大堆定义）

抽象语法树的方法主要包含 Dump() 生成代码和 translate() 翻译抽象语法树两个部分。Dump 部分比较简单，就是将一个格式直接打印出来，相对而言 translate 比较复杂。

该部分基本都是不需要思考的写写写，只不过工程量比较大。

3.2.2 初始化列表的处理

该编译器使用自底向上合并的方式处理初始化列表。前端处理完后，初始化列表存放在类型 TREE（sysyAST 的子类）中，故需要将其转化成一棵树的形式，存放到 dataCell 类型中，以方便后续处理。

每次从最内层括号开始，自左向右合并。每次遇到对应层满（查询符号管理器 VarMananger 即可获知数组形状）或者遇到右括号时，该层合并成一个更高一维的树（不满时自动填充剩下部分为空）。

需要注意的是，根据文档约定，栈上的数组未完全初始化部分需要填充零。而实现时，由于前两个阶段之前一直没有管，而 minivm 似乎对栈上内容又会自动填充零，但 qemu 不会，所以直到最后一阶段才发现此 bug。

3.2.3 sysy 到 eeyore 的转换

sysY 的 AST 转换涉及到两个问题。要将全局变量的初始化放到 main 函数的开头，要将函数内部的变量定义全部放到开头。实现上，在 defs.cpp 中实现了一个代码翻译的 buffer，当代码翻译时，遇到声明语句会存放到声明语句 eeyore AST 列表的结尾，遇到其他语句会存放到其他语句列表的结尾。在翻译完成后，再将两个语句列表合并。这样实现的好处是开销比较小，而且设置单独的 buffer 来处理，可以在实现其他模块的时候解放程序员的脑力，当 buffer 写错的时候，也比较好更改。

另外，sysY 的 continue, break 语句需要定位语法块，由于编译器实现翻译的过程是 on-the-fly，所以 sysY 会在进入子结构前实现声明好需要用到的 label，通过参数传递给子结构的 translate() 函数。具体来说，传递三个参量代表 continue 和 break 对应跳转到的 label，以及当前作用域是否是全局 (bool)。

在传递参数时，需要区分传递的是数组还是变量。例如调用 $func(a[3])$，如果形参是数组，就应该把 $a[3]$ 的地址传过去，如果形参是变量，就应该把 $a[3]$ 的值传过去。具体实现上，查询函数管理器 FuncManager 里的记录的形参列表即可获知到底是那种情况。

额外需要注意的是 or 和 and 的短路问题。这一部分在实现的时候卡了很久，对着 sort4.sy 和官方给的 eeyore 程序看了半天，我一直以为官方版本的写法太傻，我的和他明明是等价的，为何会错呢？后来用 minivm 模拟，发现类似这样的语句 $false\&\&array[-1]$ 会报访问地址错误才发现该问题。

以下是 sysyAST 中涉及到的其他方法。

atomize()	将一个表达式存放到临时变量中，并产生对应的三地址码语句
getToken()	翻译 sysY 变量对应的 eeyore 名
vectorize()	返回数组的形状，并向量化
instantialize()	将初始化列表中的常量表达式变成常值
initialize()	生成 eeyore 初始化语句
pass()	翻译传参语句

3.2.4 eeyore 到 tigger 的转换

这一部分的实现，更为简单。

在完成前面提到的寄存器分配后，大部分语句都是上下文无关的翻译，当用到的变量不在寄存器时，用事先预留的 4 个临时寄存器存放变量的值，计算完成后再恢复到栈上。

这一部分主要麻烦点在于函数调用，以及寄存器的保存。

寄存器管理器 RegManager 实现了 caller_save() 和 callee_save() 来完成对应寄存器的保存（每个函数开辟一定的栈空间，将寄存器放到栈帧上）。在函数调用、函数开头、函数返回时需要调用对应的函数。值得注意的是，如果一个变量在函数体内没有使用，则不必保存。

调用函数前，需要完成 caller-save register 的保存，接着传递参数。具体的，在 RegManager 里记录当前是第几条 Param 语句，就能得知一个变量应该通过 $\%ai$ 传参。

而函数返回时，需要将返回值存放到对应变量里。这里需要额外注意覆盖问题。

$$t = call\ f\_func$$

假设 t 存放在 $\%a1$ 中，此时应当将返回值 $\%a0$ 传递给 $\%a1$。假设 $\%a1$ 在函数调用前作为 caller-save 寄存器被保存在栈帧上，此时不应该从栈上恢复，因为 $\%a1$ 已经对应是新值。

3.2.5 符号管理器的实现

为了解决重名变量，符号管理器实现了一个日志文件。日志文件是一个栈，特别的，当前的栈顶就对应当前进入最内层作用域的记录。记录中表明了内作用域新增了那些变量，这些变量对应符号在作用域外对应哪个变量的描述符。换言之，假设当前作用域外变量名 foo 对应的变量描述符是 d_0，而当前作用域内现在新建了变量 foo，就会给内部 foo 变量分配一个新的变量描述符 d_1，并在符号表中将变量名 foo 从 d_0 重映射到 d_1，而日志文件就是记录这样的描述符对 (d_0, d_1)。

当退出一个作用域的时候，就通过日志文件将对应变量名恢复映射到外侧的变量描述符，即将 foo 恢复映射到 d_0。

这样的实现可以保证符号管理器的复杂度是线性的。另一种解决方式，是给每一个作用域都记录变量名到变量描述符的映射。但倘若如此处理，在查询一个变量信息时，需要从最内侧的作用域向外找到第一个存在该变量定义的作用域，时间复杂度不再是线性。

故，如此处理，封装性较好，同时复杂度保证是线性。

3.2.6 优化中的代码实现

因为没有作过程间分析，控制流图仅需要获得过程内部的。直接采用 bfs 的方式，比较粗暴，但因为测试样例中代码行数 n 比较小，所以 $O(n^2)$ 的复杂度仍然可以处理。

在求出 def-use 对后，常量传播、赋值传播、死代码消除、强度消减、活跃变量分析都比较直接。

循环外提的实现方式稍微麻烦一点：在 eeyore 的每个函数内检测循环体（一段代码，底部有 goto 语句跳到开头，而循环体内部不能跳出），若有常值代码则可以外提。

另外，条件固定的分支跳转指令，可以改成直接跳转。

3.3 测试

错误	发现	解决
传参未区分数组和变量	本地手写自测样例	添加函数表，记录形参信息
运算符优先级错误	priority.sy 错误	查询 wikipedia 修正
未支持短路	sort4.sy 错误 miniVM 发现 false&&array[-1]	增加短路支持
未支持单运算符 +	一直 99 分，询问同学	添加支持
寻址 *4 不对	miniVM 报对齐错误	仔细研读文档
当调用函数的返回值%a0 赋值给%a1 后，%a1 不应该再从栈中恢复	本地手写自测样例	添加更新标志,遇到标志则不用从栈中恢复寄存器
局部不完整初始化需要零填充	测试点 defn_arr.sy 报错，询问助教哥找到问题所在	添加零填充
未注意汇编指令立即数限制	测试点报 WA，盯着文档看了许久	需要判断一些立即数是否满足 int10 和 int12 的范围，若不能，需要暂存临时寄存器
取反错误，\neg x 应当是!x 而不能写成 ~x	测试点报 WA，本地压力测试找出问题	修改成!x

3.4 发现课程所使用工具存在的问题

- lex 和 yacc 对于 c++ 的一些支持不是太完善。很多时候，我们希望转换出的结构是 union 形式，因为有一些量就是常整数、有一些是实际的抽象语法树节点。而 union 中不能包含 string，这一点非常麻烦。
- miniVM 有行为缺陷：其不应该自动给栈帧刷零，这一点和 qemu 的行为不一致，导致了一些 debug 上的困难。
- eeyore/tigger 的设计有些点不是太连贯：内存地址 *4 与否比较绕；应该在 tigger 阶段就限定二元运算的第二个字节必须是 int12，且运算数 2 是数字的运算只能是 + 和 <；应该在 eeyore 阶段就允许使用位运算；对于一些优化不太方便，例如循环体内的 $a[i][j]$（i 是不变量），应当允许先计算出 $a[i]$ 的地址，这样后期常值运算强度消减比较好做，而我顺着 eeyore/tigger 的架构写完之后，发现针对数组寻址的优化不是太好改，因为 eeyore 鼓励把他变成 $a[(i*n+j)*4]$ 的形式，后续想要优化就比较麻烦。
- 评测环境：应该把性能测试和功能测试的反馈分开。当前评测平台遇到超时的情况，会直接报整体的运行超时，而没有每个点的信息。而我改完最后一个第三阶段的 bug 后，总是给我报超时，我一直以为是 bug 没改对，后来才知道是功能测试已经通过，而我的程序对 stoptime 支持不完善，在性能测试阶段超时。故产生误解。

4 实习总结

4.1 收获与体会

第一次写这么长的代码，非常有成就感！学会了写多态（2333），学会了用 github。

实习部分的难点主要是 debug （再次 2333），很多时候连着 de 了好几个 bug 发现分数纹丝未动，心态都要崩了。

编译 lab 感觉还是让大家对稍微大一点的项目不要发怵，也没什么不可战胜的。

4.2 对课程的建议

助教哥真的是太热心了！从来没见过这么好的助教，帮助非常大！实习内容上，lab 整体实现下来还是非常连贯的。

当然 tigger/eeyore 这两个 IR 可能得换一换。以及 qemu 跑起来还得单独开个 docker，我每次写代码都是用 vagrant 开个虚拟机，再套个 docker，风扇狂转，人都傻了。

老师的讲课也非常棒，前半学期比较偏编译理论，为了赶上实习的节奏，老师很快填补上了编译实习需要的知识。后半部分拓展的历史和前沿知识，也让人受益匪浅！

编译实习确实是**一门让人非常有成就感**，也能**学到知识**的课！（当然也希望是**给分好**的课！）