The report of my compiler

一年一度的编译器已接近尾声，在报告的开头首先要感谢一下辛劳的助教们，花费了很多时间协助我们完成这次 project。

本篇报告将按照大作业的 phase 进行阶段划分，总结我做的工作

Phase 1 语法分析

我们要实现的是一个部分功能的 c 语言编译器，而 c 语言的程序是无穷无尽、可长可短的，怎么识别一个程序是否是 c 语言程序呢？为此我们抽象出 c 语言的语法，用上下文无关文法进行描述，使得 c 语言程序是这个文法的一个子集，而上下文无关文法的 check 是有方法的。

本阶段的主要工作是掌握使用文法分析工具的方法，理论上来说工作量不是很大。我选择的是 antrl3.5，但不得不说 antlr3.5 确实是不太好用，在写法上稍有差异就会遇到奇怪的 bug，总之还是折腾了蛮久。但是选择 antlr3.5 的一个理由就是它的 tree rewrite 非常方便，而 phase1 的要求就是输出 AST，当时比较随意的写了些 rewrite grammar，但并没有提前预习后面需要做什么，虽然 AST 输出的像模像样，但也因此留下了后患，这个会在后文提到。

Phase 2 语义分析

在 phase1 中我们主要是用 c 语言的上下文无关文法检测一个程序是否满足要求，但这显然是不够的，比如 c 语言中用到的变量和函数都是需要事先声明的，但在上下文无关文法中我们会接受一些联系上下文就不对的程序，即语义上出现了问题，因此我们需要在 phase2 阶段进行语义分析，排除掉我们在 phase1 中会接收但实际上却有问题的程序。

（这里不得不说一下在 phase2 阶段由于我对 antlr3.5 的不信赖，我没有用它提供的 tree visitor 进行 phase2 的 coding，而是直接将 AST 提取出来自己新建一些 class 来维护树，因为我觉得这样不会再遇到和 phase1 中一样的一些奇葩的问题。但由于我在 phase1 中 rewrite grammar 时做的比较粗糙，有些信息没有表达完，所以我在 phase2 阶段对 phase1 中的 rewrite grammar 重新 re 了一遍。与此同时，由于 antlr 生成的 parser.java 中的异常处理不太好，我也对其进行了修改）

下面讲一下语义分析阶段需要处理的一些问题

c 语言可以理解为是变量与函数的定义，那么我们首先就要解决它们的定义问题。定义的东西（比如 int a）以后可能会要用到，所以需要将定义的名称（它的名字 a）与其具体的一些信息（它的类型 int）保留下来，一个很自然的想法就是用 hash 表来维护这些信息。

当然定义这方面的问题还没有结束，在 c 语言中存在作用域的问题，比如我们可以在全局定义 int a，在 main 函数的开头定义 char a，而我们在 main 函数中对 a 进行操作，是对 char a 进行操作而不是 int a。我们可以看到，不同的 scope 允许定义同样的名称，而使用则是采

取“就近原则”，而现在的关键是要处理名字的冲突问题。对于这个问题，一个很显然的想法就是，既然查询采用的是“就近原则”，那么每次遇到名字冲突的时候，在 hash 表中我们直接进行覆盖，在每当一个定义的作用域结束的时候，我们就将覆盖掉的信息还原，而我们只需要维护每个定义覆盖的那个定义是什么就可以了，而链表就是一个不错的选择。

在实现上，由于 XiaoJia 学长 Symbol.java、Table.java 写的比较清晰易懂，我就直接拿来用了。值得一提的是 Struct 和 Union 是名称定义会冲突的，但其他的定义不会与它们产生冲突，需要用不同的 Table 分别维护。

以上的问题处理好之后，就到了具体的 AST 遍历的问题了，在 check 方面无非就是按照对应的文法递归 check，当然还得返回一些必要的信息。比如 expr 类树节点，我的设计是它应当返回它是否是常数，是否是左值，它的类型是什么，如果是常数，它的具体值是什么这样的信息；而与定义有关的树节点，基本上返回值都是类型、以及其名称，用于进入对应的 Table。

在写这些的时候还是遇到了不少问题的。比如在 selfref-5090379042-jiaxiao 中 int x = f(x)，令我郁闷了好久，这个程序主要是想告诉我们在什么地方将定义的变量加入 Table 是非常有讲究的，于是乎我将变量塞入 Table 的地方改到了 init_declarator 的 initializer 之前；另外一个就是类型转换的问题，开始我随手敲了几个特判以为就差不多了，但在众多数据的教育之下，我发现我真是太年轻了，比如指针和整数间的二元运算以及指针与其他类型的 blabla；最后一个 debug 特别久的是左值的问题，也是对着 CE 中的程序特判特判，虽然在 phase2 测试中过掉了全部数据，但在后阶段的编写过程中，还发现了关于左值的一些问题，使得我对自己编译器的正确性深感怀疑。不过现在想来虽然特判这些东西在写的时候比较麻烦，但最终也能收获到许多平时几乎不用的写法，使我们对于至少是 c 语言的语言特性有了不少新的认识，也并不完全是什么坏事。

由于实际写 phase2 的时间并不长，比较赶，我在整体架构设计的并没有我预期的那么好，一不小心就很容易写错，导致在 debug 时期比较尴尬，但好歹发挥了多年 OI 和 ACM 的水平，挽救了我架构设计的失败。在 phase3 初期我还萌生过要 re 一下架构的想法，但终究败给工程量有点大，不如就这样凑合一下吧的观念继续工作了。

Phase 3 中间代码

实际上进行完 phase2 的工作之后，是可以直接将原程序翻译成 mips 代码的，但考虑到编译器的普遍写法、优化、debug 等因素，最好还是先将原代码翻译成中间代码，再由中间代码翻译成 mips 代码。对我来说翻译成中间代码最大的好处就是可以比较方便的看一些简短程序的中间代码输出来判断我写对没有，相比较而言，mips 代码稍微难懂一些。

在开始阶段，就有某位同学向我布教：Knuth 曾说过“过早的优化是万恶之源”。我觉得这句话还是蛮靠谱的，于是中间代码一直写的非常耿直，一步一步是什么就翻译成什么；而与我同寝的某位同学在写中间代码的时候，就这里搞个什么小优化，那里怎么怎么弄一下，感觉没有什么必要，毕竟本阶段保证正确性是第一要义。

具体的四元式感觉都是大同小异，我设计的有 BinOp 表示运算表达式，Branch 和 Jump 用于跳转，Call 用于调用函数，此外还是 Label、Move 和 Return 等；而四元式中的具体运算

单元，有 TempOprand 表示寄存器，有 Mem 表示内存操作，还是 Label 和 Const。我实现四元式的地方是和 phase2 一起的，即第一遍遍历树，感觉这样比较方便，比如可以直接使用 phase2 阶段的 Table。

在写中间代码的过程中我遇到了一个神奇的问题。在数组、struct 等的实现上，我一直是使用一个寄存器来追踪地址，而其他的基本元素（全局变量除外）一般都是直接将值存入寄存器中，而在写着写着的过程中，我发现对于 if (a[i]) 和 if (b) 这样的语句，括号中都会有一个寄存器，但左边的寄存器是追踪的 a[i]的地址，而右边的寄存器记录的是 b 的值，但最终我们需要返回的都是值，而我最开始的架构并不能很好的区分这样的情况，纠结了一会。经过和某位很厉害的同学的讨论并思考了一阵子后，我觉得可以在寄存器中加入一个变量来记录这个寄存器到底记录的是值还是地址，这样既能解决上述问题，需要改动的地方也相对来说较少。这之后我也发现不少同学遇到了这个问题，当然我也将这个写法分享给了他们。

大概快到 code review 的时候我开始写无限寄存器版本的 mips 代码输出。参考了助教的意见，将全局变量额外处理，每次随用随取，用完即存，于是将全局变量声明在.data 区域中，实际上对全局变量的操作就和上面一段对数组的操作差不多，都是用在.data 中声明的 label 进行访问。而对于无限寄存器的分配，由于写法上会涉及到例如$sp 之类的寄存器，我们称之为有名寄存器，而其他的中间寄存器称为无名寄存器，我仅对无名寄存器采用动态分配，这样的写法使得在 phase4 后的寄存器分配时，只需要将分配到寄存器的中间寄存器更改为有名寄存器即可，工作量比较小

当时终于在 deadline 那天的下午把八皇后跑出来了，非常开心的去 code review 了，但由于我赶的比较急，mips 代码输出是直接在原来四元式输出的函数中直接改的，金天行助教在 code review 中要我输出中间代码的时候我就囧了，于是就这样无情的被他扣了 0.5 分。但我现在觉得虽然扣这 0.5 分对我分数影响确实可能不大，但确实对我的一种警醒：在做一个大 project 时需要对整体架构有所把握，新增和修改内容需要想清楚，更不能因为做到了后面而轻易就把原来的东西给覆盖掉了。

Phase 4 寄存器分配和优化

本阶段首先需要确认每个寄存器的活性范围，然后可以通过其活性范围进行寄存器分配，并做适当的优化

活性分析我参考了虎书，如果一个变量被使用了，那么它一定在之前被定义过，一直向前沿各种可能找到它的定义，沿途都是这个变量的活性范围。

虎书中的描述如下：

活跃信息（入口活跃信息和出口活跃信息）可以用如下方式从 *use* 和 *def* 求出：

（1）如果一个变量属于 *use*[*n*]，那么它在结点 *n* 是入口活跃的。即如果一条语句使用了一个变量，则该变量在这条语句入口是活跃的。

（2）如果一个变量在结点 *n* 是入口活跃的，那么它在所有属于 *pred*[*n*]的结点 *m* 中都是出

口活跃的。

（3）如果一个变量在结点 n 是出口活跃的，而且不属于 $def[n]$，则该变量在结点 n 是入口活跃的。即，如果变量 a 的值在语句 n 结束后还需使用，但是 n 并没有对 a 赋值，则 a 的值在进入 n 的入口时就是需要使用的。

考虑到效率的问题，由于大部分时候每条语句的前驱都是它的前一条，即使不是也一般在它前面，所以从后往前 for 每条语句是最合理的。我写完按每条语句作为一个节点的活性分析后，发现大部分数据都跑的很快，所以没有再划分 block。

做完活性分析后，我采用 linear scan 进行寄存器分配，具体实现是对于每个变量，考虑其在活性分析中出现最早和最晚的位置，得到一个区间，对于所有区间按左端点从小到大扫描进行分配，若有 spill 的情况，则踢掉区间右端点最大的那个变量。

实际上 linear scan 效果还不错，详细见下表

	分配前	分配后	优化比例
basicopt1-hetianxing.c	2920809	578880	0.8018083
board-5100379110-daibo.c	27739	17818	0.3576553
Bulgarian_solitaire-5110379024-wuhang.c	2102036	974492	0.5364057
expr-5110309085-jintianxing.c	117716	44317	0.6235261
hanoi2-5100309153-yanghuan.c	769899	671608	0.1276674
hanoi-5100379110-daibo.c	276776	245050	0.114627
hashmap-5100309127-hetianxing.c	658164	337232	0.4876171
heapsort-5100379110-daibo.c	32500665	13024384	0.5992579
horse2-5100309153-yanghuan.c	17196143	8868090	0.4842977
horse3-5100309153-yanghuan.c	24007464	13163940	0.451673
horse-5100309153-yanghuan.c	19208531	11004245	0.4271168
hpplus-5100309153-yanghuan.c	6055	2917	0.5182494
magic-5100309153-yanghuan.c	3239598	1500036	0.5369685
maxflow-5100379110-daibo.c	33435044	15013464	0.5509662
multiarray-5100309153-yanghuan.c	11449	5619	0.5092148
nqueencon-5100379110-daibo.c	15527	9029	0.4184968
pi-5090379042-jiaxiao.c	18410687	7092053	0.6147861
prime2-5100309153-yanghuan.c	8730967	4125923	0.527438
prime-5100309153-yanghuan.c	8758015	4160699	0.5249267
qsort-5100379110-daibo.c	5425704	2456492	0.5472492
queenbitwise-5110379024-wuhang.c	229200	127353	0.4443586
queens-5100379110-daibo.c	1974603	902242	0.5430768
spill2-5100379110-daibo.c	15683	12313	0.2148824
struct5-5110379024-wuhang.c	125674	58794	0.5321705
superloop-5090379042-jiaxiao.c	5306505	2592981	0.511358
tak-5090379042-jiaxiao.c	2425358	2061555	0.1499997
treap-5110309085-jintianxing.c	11244238	6220404	0.4467919
twinprime-5090379042-jiaxiao.c	2682474	1164888	0.5657412
varptr-5100309127-hetianxing.c	7698	4620	0.3998441

（注：所有表格都将指令数较少的数据移除）

其中优化比例较大的，如 basicopt1 和 pi 等，都是频繁的使用局部变量，这也与我全局变量的处理方式有关

接下来我开始进行优化，首先是死代码消除，比如你在 c 程序中写了一句 a=1，但是之后再也没有用过 a，这句话实际上并没有意义，去掉之后对结构没有任何影响。借用活性分析的结果，只要某条语句 def 的变量没有出现在其 out 集合中，则这句话就可以直接删掉，详细指令数变化见下表

	优化前	优化后	优化比例
spill2-5100379110-daibo.c	12313	9748	0.208316414
basicopt1-hetianxing.c	578880	473836	0.181460752
nqueencon-5100379110-daibo.c	9029	7487	0.170783032
qsort-5100379110-daibo.c	2456492	2103776	0.143585243
hpplus-5100309153-yanghuan.c	2917	2500	0.142955091
hashmap-5100309127-hetianxing.c	337232	295832	0.12276415
multiarray-5100309153-yanghuan.c	5619	4947	0.119594234
twinprime-5090379042-jiaxiao.c	1164888	1032230	0.113880476
varptr-5100309127-hetianxing.c	4620	4128	0.106493506
queens-5100379110-daibo.c	902242	839824	0.069180996
maxflow-5100379110-daibo.c	15013464	14174464	0.055883173
superloop-5090379042-jiaxiao.c	2592981	2479564	0.043740004
board-5100379110-daibo.c	17818	17061	0.042485127
queenbitwise-5110379024-wuhang.c	127353	123044	0.033835088
magic-5100309153-yanghuan.c	1500036	1449583	0.033634526
Bulgarian_solitaire-5110379024-wuhang.c	974492	944144	0.03114238
horse3-5100309153-yanghuan.c	13163940	12784149	0.028850861
treap-5110309085-jintianxing.c	6220404	6044381	0.02829768
hanoi2-5100309153-yanghuan.c	671608	655228	0.024389227
hanoi-5100379110-daibo.c	245050	240953	0.016719037
horse-5100309153-yanghuan.c	11004245	10824232	0.016358505
struct5-5110379024-wuhang.c	58794	58065	0.012399224
prime-5100309153-yanghuan.c	4160699	4131283	0.007069966
heapsort-5100379110-daibo.c	13024384	12954376	0.005375149
prime2-5100309153-yanghuan.c	4125923	4108099	0.004320003
pi-5090379042-jiaxiao.c	7092053	7091651	5.66832E-05
expr-5110309085-jintianxing.c	44317	44315	4.51294E-05
horse2-5100309153-yanghuan.c	8868090	8867876	2.41315E-05
tak-5090379042-jiaxiao.c	2061555	2061551	1.94028E-06

结果按照优化比例降序排序，可以看出优化幅度并不明显，仅有几个程序优化幅度超过 10%，但实际上其他很多优化都依赖着死代码消除。

接下来我进行了窥孔优化，例如处理掉 sne，seq 这样的伪指令（在 spim 中会被翻译成多条语句，由于在四元式阶段实现的不太好，一般会出现 seq t1, t2, t3、beq t1, $0 label，将其改成 bne t2, t3, lable 将节省一些指令数），还在里面写了一些传播等（比如将 li t1, 1、add t2, t3, t1 改写为 add t2, t3, 1 等），但效果一般，没有达到我的预期，详见下表

	优化前	优化后	优化比例
superloop-5090379042-jiaxiao.c	2479564	1246009	0.497488671
hashmap-5100309127-hetianxing.c	295832	235827	0.202834717
queenbitwise-5110379024-wuhang.c	123044	100691	0.181666721
treap-5110309085-jintianxing.c	6044381	5037565	0.166570572
expr-5110309085-jintianxing.c	44315	36949	0.166219113
queens-5100379110-daibo.c	839824	712714	0.151353141
horse2-5100309153-yanghuan.c	8867876	7628054	0.13981048
prime2-5100309153-yanghuan.c	4108099	3534399	0.139650968
maxflow-5100379110-daibo.c	14174464	12220134	0.137876818
varptr-5100309127-hetianxing.c	4128	3587	0.131056202
prime-5100309153-yanghuan.c	4131283	3614043	0.125200815
horse-5100309153-yanghuan.c	10824232	9504450	0.121928466
basicopt1-hetianxing.c	473836	417178	0.119573017
magic-5100309153-yanghuan.c	1449583	1282726	0.115106896
horse3-5100309153-yanghuan.c	12784149	11334569	0.113388854
multiarray-5100309153-yanghuan.c	4947	4398	0.110976349
hpplus-5100309153-yanghuan.c	2500	2260	0.096
board-5100379110-daibo.c	17061	15429	0.095656761
Bulgarian_solitaire-5110379024-wuhang.c	944144	862099	0.086898821
nqueencon-5100379110-daibo.c	7487	6858	0.084012288
tak-5090379042-jiaxiao.c	2061551	1894811	0.080880851
heapsort-5100379110-daibo.c	12954376	12326625	0.048458606
struct5-5110379024-wuhang.c	58065	55869	0.037819685
hanoi-5100379110-daibo.c	240953	235324	0.023361402
qsort-5100379110-daibo.c	2103776	2057865	0.021823141
hanoi2-5100309153-yanghuan.c	655228	642942	0.018750725
twinprime-5090379042-jiaxiao.c	1032230	1028449	0.003662943
pi-5090379042-jiaxiao.c	7091651	7088041	0.000509049
spill2-5100379110-daibo.c	9748	9748	0

结果是按照优化比例降序排序的，不过可以看到也有大部分优化比例在 10%以上，说明至少优化还是有一定效果的

小结

这次写编译器，我一个非常大的收获是做一个 project，最核心的是设计一个好的架构。一个好的架构，意味着更好的兼容性，更方便的 debug，也不容易出现需要大规模改动的地方。

一开始的时候我对写编译器这个东西还有点抵触，什么都不太懂，觉得写起来也没什么意思,但我在临近尾声的时候,反而觉得写这样一个project的意义还是非常大的。看邮件好像以后要将编译器弄成每个阶段都毫不相关，我觉得这有利有弊吧，但我个人觉得还是不要那样比较好，弄成每个阶段不相关的话，最终会让大家失去对于一个工程整体架构把握的意识。

最后再次感谢助教们，感谢你们对我们的帮助，感谢你们辛劳的付出。