Report for Compiler 2015

魏楚

2015 年 5 月 31 日

1 简介

众所周知，写编译器是一件比较困难的事情。即使现在编译器技术已经很成熟了，从零开始写一个支持某种主流语言的主要特性的编译器依然不是那么简单。我耗时两个月，用C语言完成了一个简化版C语言编译器。我将在本文中叙述写编译器的具体方法，以及过程中的种种艰难苦恨，希望能够分享一些经验，让学弟学妹们少走点弯路。

写编译器大致上可以分为四个阶段：

1. 语法分析（Phase1）
2. 语义检查（Phase2）
3. 生成中间代码（Phase3）
4. 优化与生成MIPS代码（Phase4与Phase5）

本文将会依次介绍这四个阶段，余下的部分讲述我实现的bonus项目。

注意：我写的代码全都存于phase5/目录下，为了评测方便将一部分代码复制到了根目录下，而与最终评测无关的代码并没有复制进去。

2 语法分析

语法分析可以分为词法分析和语法分析两部分，这一阶段有许多工具可以用，如yacc之类的，可以自行搜索相关的资料。我手写了词法分析器与语法分析器。

词法分析就是把源代码拆成一个一个的token（如变量，符号，数字等），并去掉注释，这一步非常简单，我采用了暴力分析法。相关代码有removeComments.c/.h, split.c/.h, tokens.c/.h

语法分析就是根据某个（给定的）上下文无关文法，将token组织成一棵抽象语法树。这一步比较困难，我参考了龙书与虎书的相关内容，完成了LL(1) 的语法分析器。

开始我打算写最简单的LL(1)语法分析器，不过发现需要实现的编译器对应的上下文无关文法既有左递归也有左公因子，而且不知道怎么消除它们，网上到处都搜不到切实可用的算法。于是我继续研究LR(1)语法分析器，写了一部分之后发现生成的表格太大，可能内存不够。之后我又研究了LALR(1)语法分析器，不过写的过程中发现实现起来比较困难，于是最终又转回了LL(1)分析器。

幸运的是，该文法是可以转化为LL(1)文法的，转化方式龙书上也写了，不过要写成程序不是很方便。为了获取LL(1)文法，我先手动去除了左递归，之后花了许多时间手动消除左公因子。由于其中有些生成式的间接左公因子藏得特别深，消除过程非常麻烦，不过最后还是完成了，尽管显得非常丑陋。为了区分终结符与非终结符，我给每个终结符加了一对单引号，而空字符ϵ写成'__'。

有了LL(1)文法之后LL(1)文法分析器就很好写了，可以参考龙书中的设计，其核心在于FIRST与FOLLOW集合的计算。于是，我们可以将输入的源码转化为具体语法树(CST)了。我的LL(1)分析器对应文件LL.c和LL.h，以grammar1.in作为输入，最终生成文件grammar.out，包括了所有语法规则与转移表。不过开始时我的可执行文件每次都需要读取文件grammar.out，为了方便，我又另外写了grammar_gen.c把grammar.out改造为头文件grammar.h，这样编译之后的可执行文件就可以单独运行了。

之后需要进一步将CST转化为抽象语法树(AST)，由于我的CST非常丑陋，因此花了许多时间才完成这一步。我采用了廖超助教的架构（https://github.com/zeonsgtr/compiler2015/tree/master/framework/ast），定义了一个AstNode结构体，其中type表示节点类型，data是其对应的字符串，c,num,cap一起实现了一个变长数组，其他成员是后期添加的，本阶段用不到。这一步对应到parser.h 和parser.c

3 语义检查

做完语法分析就是做语义检查了，这一步主要是判断代码是否有编译错误，同时提供一部分信息给以后的阶段使用。

这一阶段需要构造两个符号表，前者记录struct/union名字，后者记录变量/函数名。我手写了hash表，对应hash.c和hash.h。核心代码是semantic.h和semantic.c。

要想写好这一块，需要考虑如何表示“类型”。除了声明变量时安排类型外，还需要给每个（子）表达式安排类型。我用retType域表示表达式的类型，由于C语言没有垃圾回收机制，为了防止出现两个指针指向同一个位置导致回收出错，我精心设计了retType。一般是新增一个节点放到表达式的c域里，然后把retType指向它，有些地方这个节点的c需要指向其他节点，为了防止两次free，我将此节点的num置为0。（见addAno函数）

之后就是检查类型匹配，变量是否声明前使用，break/continue是否在循环内，左值与右值等细节。

此外，我还在这个阶段把字符串常量转义了，也把一些常量表达式计算了出来，同时也把类型的大小算了出来，为下一阶段做准备。

4 生成中间代码

生成中间代码(IR)是编译器前端的最后一步，需要根据AST生成语言无关的中间代码，方便最终生成汇编码。

我依然参考了廖超助教的架构（https://github.com/zeonsgtr/compiler2015/tree/master/framework/ir），定义了Object, Sentence, Function等多种结构体，相关代码为ir.c和ir.h

中间代码的设计很关键，因为它直接影响了之后的优化和生成MIPS代码。我在这一阶段走了不少弯路，多次推掉重写，大大影响了进度。

我的IR由若干的Function组成，其中$start$函数为虚拟函数，保存了全局变量等信息，3个内置函数printf,malloc,getchar留到下一阶段实现。初始化写成赋值语句塞到对应的函数里面。每一个Function由两个ObjectList和一个SentenceList组成，分别是参数列表，变量列表和语句列表。我假设有无限的寄存器，每一个变量都赋一个寄存器给它。每个寄存器有一个pd值，表示它是常量，指针（存着目标地址）还是内存块（存着数据）。特别需要注意的是struct与union 不同，数组与struct 不同，数组与指针不同等等。

比较麻烦的一点是这一阶段不太容易发现bug，直到准备生成MIPS代码时才发现架构不利于生成代码，于是只能重写。

这一阶段，我还维护了一些与函数调用相关的信息，方便下一阶段的工作。

5 优化与生成MIPS代码

这是整个编译器的最后一步，也是不那么容易的一步。

只要IR写得对，那么生成MIPS是比较自然的，不过也需要注意一些细节，同时也要搞清楚函数调用应该怎么分配栈空间。

我试图先写三寄存器的MIPS通过正确性，之后再写寄存器分配以大幅减少指令数，然而因为调试正确性用了好多天，已经来不及做寄存器分配了，于是只能在三寄存器的基础上做了一些优化。

优化可以优化AST（如利用AST做公共子表达式消除），优化中间代码以及优化最终代码，我主要优化了中间代码和最终代码。为了方便修改MIPS代码，我在生成阶段先用sprintf函数输出到一个缓冲区里面，之后进行一些优化，最后再真正地输出。以下是我做的一部分优化：

1. 3个内置函数都是手写的，尤其是printf函数经过了精心设计，尽量减少指令数。

2. 做了一些函数结构优化，比如不调用函数的函数（Leaf Function）不保存$ra的值，返回值小于4位就用$v0而不是栈空间来传递等。

3. 做了一些窥孔优化，如涉及常量的运算尽量不用li存到寄存器里而是用立即数，删除不必要的jump等。

这一阶段对应的代码是translate.c和translate.h，同时也不停地修改ir.c和ir.h

经过以上几点优化，我的三寄存器代码目前可以通过5个门限，和写得比较差的寄存器分配结果差不多。

6 Bonus项目详解

6.1 handwritten lexer and parser

6.1.1 使用方法

进入phase5/目录里，执行make -f makefile_ast之后生成可执行文件astPrint，输入一段C语言源码即可输出其对应的AST：./astPrint < source.c > ast.txt

6.1.2 实现方式

见第二节语法分析

6.2 pretty printer

6.2.1 使用方法

进入phase5/目录里，执行make -f makefile_pretty之后生成可执行文件pretty，输入一段C语言源码即可输出它的美化版本

6.2.2 实现方式

pretty printer看似很好写，真正写起来就会发现事情并不是那么单纯。首先，为了尽量保留原程序的东西，不能在AST的基础上进行工作，因为括号等信息会丢失。而假如只进行词法分析的话很难区分一个token的意义。比如同样是*，它可能是解引用符，也可能是乘号；即使是同一个符号也有不同的缩进方案，如分号在for语句的括号里不换行，而在其他地方一般是要换行的。

最终我选择了在CST基础上进行构造，不过由于我的CST非常丑陋，所以工作量还是比较大的，需要注意许多细节。缩进方式参考了我平时的缩进风格。

6.2.3 注意事项

CE的代码会返回Compile Error信息，注释会被清空，因为我认为这两种情况都是没有办法美化的。同样地，pretty也需要用到parser.h 里面的接口。

6.3 using C programming language

这个貌似没什么好讲的。

那就吐槽一下吧。用C语言写编译器最麻烦的地方在于不能面向对象，同时也不像C++/JAVA那样有一些比较好用的标准库。所以hash表要手写，建AST和IR 的时候想尽办法用一个struct表示所有类型的节点，等等。整个过程还是比较痛苦的，有时候会觉得用C语言写编译器得不偿失。

7 小结

编译器是我有史以来写过的最大的Project，总代码量几千行，而且难度也是最大的。在此期间，我也发现了自己的一些不良编程习惯，如写代码之前没有考虑好架构。在此期间，我不断地遇到困难又不断地战胜它，比如语法分析器多次推掉重写，手动去除左公因子和左递归各种出错和郁闷，语义分析出现各种奇怪的bug，中间代码多次重新设计，最后3个数据点的正确性足足调了一天才成功，写了各种各样的优化终于卡过了几个门限。不过最后，看到自己的编译器能够顺利地跑起来，还是感到由衷的欣慰的。

如果说有什么后悔的，那就是之前写代码的时候效率不够高，有时候会高估自己的战斗力。虽然总的来说并没有等到最后才开动，但是平时写的时候总是进展速度缓慢，而且犯错率高，于是只能在deadline临近之际熬夜狂写。

郑重地给学弟学妹留下三句话：

1. 不要拖延
2. 不要拖延
3. 不要拖延

最后，我要感谢各位助教的辛勤劳动，如廖超助教的AST架构和IR架构给了我很大的帮助，蒋舜宁助教通过提前测试帮助我们熟悉评测环境。