编译技术课程设计 – 申优文档

游子诺

本学期编译器课程设计的就要接近尾声了，在过去的 3 个多月时间里，我相继完成了文法理解、词法解析、文法解析、错误处理、中间及目标代码生成、生成优化六个循序递进的问题。和计组一样再一次地通过不断迭代和优化自己的设计以实现更丰富的功能，不过与计组课程设计不同的是，编译课设更加让我体会到了“设计”方面的提升，虽然理论性的编译文法分析、解析和代码生成是给出的，但是具体到教程组所给定的类 C 文法和 MIPS 目标代码，格式各样的问题开始接连显现，这就需要我们自己完成计组中细致而具体的设计工作，从而在代码层面能够实现，下面我将依次针对每个阶段中自己遇到的典型问题进行总结和反思。

整体工程结构分析

语言与编程思想：借用理论课上老师所说的话，课设所需要我们实现的编译器整体是一个一遍式编译器，在优化环节时可能会带有对中间代码的多遍运行，因此反映在 main 函数中其实就是流水线顺序式的功能部件逐个运行，这虽然简单看是一种顺序式流程，但是却具有明显的模块化与封装特点，通过面向对象的思想，设计好各模块之间的接口关系，从而解耦之间的功能，况且在编译器实际运行中还存在着许多数量未知的存储与映射关系，使用基础的 C 语言实现动态容量控制和泛型的难度较大，而这些问题在 C++的 STL 容器（如 vector、map 和 set）中都能很好地解决，因此在语言选择时我使用 C++语言进行编程，但由于初次使用 C++进行面向对象的编程，因此回顾看来很多 C++高级设计特性并没有有效地利用起来，也部分加重了重写代码的负担，因此建议以后初次使用语言实现面向对象编程时，除了简单看看菜鸟教程外，还是应该动手实践几类基本情况或翻看一个小型成熟的工程文件代码。

调试日志与 Log 文件：授人以鱼不如授人以渔，在编译过程中确实遇到了很多细节性的坑点，但是这些 bugs 常常因人而异，因此**更普适的是总结 debug 的方法**，将编译器的中间运行结果记录下来形成 log 文件就是重要一例，其能够帮助你更快地修复深层次的逻辑问题。像在使用 Python 进行实验任务时，其现有的 info 包能够支持各类日志信息的写入。在

C++中我没有找到很合适的，因此自定了 Debug 类，在类中实现了各种静态方法进行不同调试信息的写入，例如以 DEBUG 开头的方法通过 ofstream 将写入相应的文本日志文件，而以 WARNING 和 ERROR 开头的方法则直接在命令行上进行显示，以及时显示在动态检查时所存在的不期望行为。当然，还有一种调试设计我认为也非常好，那就是使用 C++库中所含有的<cassert>进行检查，及时在运行中发现不正确行为。以下是部分日志文件片段：

```
一部分递归下降解析函数过程
                        >>> <语句>
                         >>> <写语句>
                          @ printf PRINTFTK
                          @ ( LPARENT
                             >>> <字符串>
                              @ 5 !=    STRCON
                             <<< <字符串>
                          @ , COMMA
                             >>> <表达式>
                                 >>> <项>
                                     >>> <因子>
                                         >>> <标识符>
                                          @ n IDENFR
                                         <<< <标识符>
                                     <<< <因子>
                                 <<< <项>
                             <<< <表达式>
                          @ ) RPARENT
                         <<< <写语句>
                       @ ; SEMICN
                      <<< <语句>
```

```
函数变量的栈分配偏移与大小情况
[Function Alloc] k@complete_flower_num -> offset = 0, size = 512
[Function Alloc] i@complete_flower_num -> offset = 512, size = 4
[Function Alloc] j@complete_flower_num -> offset = 516, size = 4
[Function Alloc] n@complete_flower_num -> offset = 520, size = 4
[Function Alloc] s@complete_flower_num -> offset = 524, size = 4
[Function Alloc] x1@complete_flower_num -> offset = 528, size = 4
[Function Alloc] y@complete_flower_num -> offset = 532, size = 4
```

```
基本块划分与活跃变量分析
>>>> Block No. 15
PRE { 13, }
NEXT{ 16, }
DEF{ k@complete_flower_num, }
USE{ n@complete_flower_num, i@complete_flower_num, }
IN{ n@complete_flower_num, s@complete_flower_num, j@complete_flower_num, i@complete_flower_num, }
OUT{ n@complete_flower_num, s@complete_flower_num, j@complete_flower_num, k@complete_flower_num, i@complete_flower_num, }
complete_flower_num_else1:
k@complete_flower_num[n@complete_flower_num] = i@complete_flower_num
```

```
全局寄存器分配
[DEBUG] @factorial: $s7 <=== Name: n@factorial Weight: 3
[DEBUG] @factorial: X    <=== Name: 1 Weight: 1
[DEBUG] @factorial: X    <=== Name: t4@factorial Weight: 1
[DEBUG] @factorial: X    <=== Name: t5@factorial Weight: 1
[DEBUG] @factorial: X    <=== Name: t6@factorial Weight: 1
```

开发与文件组织：由于缺乏 C++工程的开发经验，因此最初我对头文件和主文件的组织做得并不是很好，而且常常出现两个头文件交叉引用的不合理情况，后来逐渐熟悉后才相对有所改进，逐渐将头文件设计与 Java 中的接口设计相结合，引导和帮助自己在考虑具体实现前抽象地设计每个对象的行为，最后才填充 cpp 文件具体实现功能。当然，我也从室友完成编程过程中了解到一种 hpp 文件的 c++文件形式，其比较 Python 的文件，将声明和实现放在同一个文件中，后来的同学也可以权衡一下这几种组织方法选择合适的。最后，我也希望课程组在评测机上能够关注对 C++工程组织的实现能力，目前所采用的 VS2019 所生成的代码文件完全没有依托文件夹进行文件组织，希望能够考察如使用 Cmake 等工具进行编译文件的能力。

以上就是我在编译课设与整体工程开发有关的心得与体会，希望能够后来的读者带来启发。

任务一 —— 文法解读与词法分析

这学期的课设少了以往的 Pascal 编译器源代码阅读和编译器设计，更多是比较直接的代码任务。该部分任务比较轻松，只是注意尽量对该部分进行有效的封装，以便能够为后续

的文法解析提供支持。

任务二 —— 语法分析

相较于词法分析，语法分析的难度高了一些，在文法不严格满足 LL(1)的情况下，我们需要采用偷窥更多个 token 的方法来决定递归下降时的入口函数。同时，如果仅靠语法结构，我们还会遇到语法完全一致的<有返回值的函数调用语句>和<无返回值的函数调用语句>，这时就需要使用额外的结构记录函数声明时的信息，在这里小结两点：

1. 增加 try-catch 结构和可视化良好的递归下降结构：虽然本次任务不考虑异常情况，程序理想运行，但实际中若出现带回溯的递归下降解析、编译异常等情况，预留好的 try-catch 接口就能提供更好的处理。
2. 可尝试开始考虑设计（甚至搭建）符号表，并为符号表填充预留合理的函数结构：这是我在课设中吃的比较大的亏，前进入目标代码生成前我基本都已满足目前需求为准，基本将程序写成了统一和工程化的字符串匹配，但在后期不同类型的语言结构对应不同的制导动作，因此大量工作需要重做，还引入了 bug 风险。

任务三 —— 错误处理与语义分析

该部分考察了几种简单基础的代码错误，并且需要编译器能够支持错误的局部化处理。要通过该部分的测试比较轻松，但更重要的是要开始自主建立设计符号表，结合之后的中间代码生成，在此简单介绍一下关于符号表的处理方式：

1. 符号表组织结构：不用于教材上的结构，在此使用树状结构维护符号表（仅由全局结点和函数节点两节组成），支持一定程度的树状扩展。
2. 符号表树上的每个结点均保存该块中的局部符号项，包括：const（常量）、variable（变量）、function（函数头）、label（标签）、temp_normal（临时变量）、temp_const（临时常量）、temp_string（字符串）。
3. 在符号表维护的增删查改时，对应有以下 OO 思想：
   a) 工厂模式：避免在函数各部分使用 new 符号项，同时避免创建错误。
   b) 封装：将符号树封装成对象，通过特定的方法向其中插入和查询。

任务四 —— 中间代码与目标代码生成

在该部分，中间代码的设计与目标代码生成框架将对后期编译优化产生重大影响，我一开始拿到中间代码设计要求时一头雾水，感觉提示太少，自由度太大，好在阅读了前面几届学长写的博客和开源的代码，我开始尝试构建自己的中间码规范。我大致花了 2 天设计和反思我的中间码和符号项设计，而后快速地动手实现出来。在我看来，设计这一步无论对实现速度还是后期扩展都有很大帮助，不求形式和美观，跳出代码在稿纸或注释中实现出来即可。

```
/* 除数值以外，其余均为SymbolItem类，f - function, c - const, v - variable, t - temp, l - label */
/* 四元式类型 ------------ 解释 ---------------------- 操作数A ----------- 操作数B ------------ 结果数 */
enum QuaterType
{
    Undefined,
    // 声明
    VarDeclar,      //   声明变量                  None            None            v
    ConstDeclar,    //   常量声明                  None            None            c
    // 函数
    FuncDeclar,     //   函数声明                  None            None            f
    FuncParaDeclar, //   函数变量声明              None            None            v
    FuncParaPush,   //   函数变量推入              v/c/t           None            None
    FuncCall,       //   函数调用                  f               None            None
    AssignRet,      //   返回值赋值                None            None            v/t
    FuncRet,        //   函数返回                  (v/c/t)         None            None
    // 算术
    Add,            //   算术加法                  v/c/t           v/c/t           v/t
    Sub,            //   算术减法                  (v/c/t)         v/c/t           v/t
    Mult,           //   算术乘法                  v/c/t           v/c/t           v/t
    Div,            //   算术除法                  v/c/t           v/c/t           v/t
    Assign,         //   赋值                      v/c/t           None            v/t
    // 条件
    EqlCmp,         //   等值比较                  v/c/t           v/c/t           v/t
    NeqCmp,         //   不等比较                  v/c/t           v/c/t           v/t
    GtCmp,          //   大于比较                  v/c/t           v/c/t           v/t
    GeqCmp,         //   大于等于比较              v/c/t           v/c/t           v/t
    LtCmp,          //   小于比较                  v/c/t           v/c/t           v/t
    LeqCmp,         //   小于等于比较              v/c/t           v/c/t           v/t
    // 分支
    Goto,           //   无条件跳转                l               None            None
    Bnz,            //   有条件跳转(~0)            l               v/c/t           None
    Bz,             //   有条件跳转(0)             l               v/c/t           None
    // 标签
    SetLabel,       //   设置标签                  l               None            None
    // 数组
    ArrayQuery,     //   根据索引取值              v               v/c/t           v/t
    AssignArray,    //   数组赋值                  v/c/t           v/c/t(indx)     v
    // IO
```

图 1 代码注释中的中间码设计

同时，对于设计合理性，我的中间码设计在现在看来还是不错的，因为中间码中保存的信息量很丰富（传递的都是对象的索引），这也有助于我进行后期优化时各种对象都能通过简单几次指针就能够轻松地访问到，为各种情况的优化提供可能。

而在中间码形成后，就是目标代码的生成了。在最初接触到目标代码生成时，我基本上将 MARS 当作栈式符号机进行使用，以符号树上的节点为单位进行每个变量相对运行栈$sp 指针的偏移量。对于每一步中间码的操作，均采用使用从内存中取操作数，运行或操作后存

入结果变量。在实现时，注意将中间码的翻译逻辑与 mips 代码的生成逻辑完全解耦，以方面后期进行寄存器分配时寄存器池能够隐式地生成相关 mips 代码。

任务五 —— 代码优化

代码优化是本次作业中灵活性最大，投入精力最大的地方，其中既了实现许多常见的后端优化，也基于给定的测试程序有针对性地使用了小技巧，在此分别进行简述：

内联展开:

内联展开的函数对象 —— 无自递归调用，无循环，中间代码数量不大于 50 条。在进行函数内联时，主要是考虑中间码、符号项的嵌入与函数参数、返回值的改写。

对于中间码和符号项,由于先前处理时保存的信息很丰富,因此移植起来工作量也更大，主要工作是将原本属于其他函数的符号项**无冲突**地迁移至新函数中，这其中就涉及到变量、常量、label 等各项的 ID 替换，以及其所属块信息的变更。建议实现时在原有的工厂模式中新增 InlineCopy()方法专注于解决此问题。最终内联后的中间代码效果应为：中间码对象本身应在整个序列中有且仅有出现一次，除全局量的符号项都只对应一个函数。

而对于函数参数改写与返回值改写，则是使用 Assign 四元式进行替换即可。

下图演示了一个简单函数的 inline 展开结果。

```
int inline()
para int a
para int b
para int c
t@inline = a@inline * b@inline
t1@inline = t@inline - c@inline
ret t1@inline

void main()
var int x
var int y
var int z
main_inline_inline_begin:
$a@main = 1
$b@main = 2
$c@main = 3
$t4@main = $a@main * $b@main
$t5@main = $t4@main - $c@main
t3@main = $t5@main
main_inline_inline_end:
x@main = t3@main
ret
ret
```

```
int inline(int a, int b, int c){
  return (a * b - c);
}

void main(){
  int x,y,z;
  x = inline(1,2,3);
  return;
}
```

临时寄存器分配与全局寄存器分配:

在工程架构上，我定义了寄存器池 RegisterPool 类作为寄存器的中心调度模块，中间代码翻译逻辑仅需传入其所需要的符号项，寄存器池将完全透明地进行前期的寄存器分配操作（分配寄存器并进行内存操作），传递回翻译逻辑进行其他显式的目标代码生成，在这种结构的支持下，除了不处在寄存器池中的寄存器（如$v0 作为返回值）和数组操作，其他与内存相关的操作均由寄存器池隐式的使用。当一个寄存器需要加载新的内存值时，池对应着四种操作：

- ✓ unmap - 解除该寄存器与原项的映射关系
- ✓ writeback - 将寄存器内容写回原项（可选）
- ✓ map - 建立寄存器与新项的映射关系
- ✓ load - 将新项的内存数据加载至寄存器（可选）

对于临时寄存器，我在寄存器池中采用 FIFO 算法，基于活跃变量分析中 out 集合，在每个块结束时考虑是否将临时寄寄存器的值写回。活跃变量在实现时既可以像教材上一样进行多遍数据流分析，直至不再更新位置，也可以采用图论的 BFS 算法进行更新：当某个块的 out 集合被更新时，说明其由潜力影响其前缀，加入队列。

```
while queue is not empty do
    block ← queue.pop_front()
    IN[block] ← use[block] union (OUT[block] – def[block])
    for pre in previous(block) do
        if OUT[pre] – IN[block] not empty then          // 能够更新
            OUT[pre] ← OUT[pre] union IN[BLOCK]
            queue.push(pre)
        end if
    end for
end while
```

对于全局寄存器，我使用的是引用计数，以函数为单位按循环进行加权，选择排序靠前的跨块非全局变量优先分配，不过需要注意的是，全局寄存器值的加载一定在进入函数和函数调用返回时统一执行（因此只有将高频调用函数 InLine 展开才有显著效果），而非像临时寄存器一样懒惰地加载，否则寄存器值会出现旧值错误。根据结果表明，在使用以上的寄存器分配策略后，Memory 类指令将不会成为程序运行的主要开销，ALU 为主要开销。

复制传播:

相较于教材上的标准解法，我在使用复制传播消除 Assign 赋值时主要基于块内信息和活跃变量，给定一个赋值语句 Assign A=B，其结果端 A 如果满足在后续块中不再活跃，那么可以在赋值语句后面直至下一次将 A 作为结果端之前，若该部分有 A 作为操作数的中间码，都可以替换为 B 作为操作数。

常量传播:

该部分实现比较简单，具体来说就是在进行表达式运算时，若遇到操作数均为常数的中间码，可以由编译器进行计算赋予结果数，省略该操作。

窥孔优化:

1. 粘合 cmp 类操作和 bz 操作，生成 ble，bge，blez，bgez 等 mips 指令。
2. 识别邻近的三条算术指令与赋值指令，当其具有“取模运算”的显著特征时，直接使用乘除法模块所支持的取模运算实现。
3. 对于操作数存在一个常数的情况，可使用 addi、subi 等 cali 型指令。

前端优化（将所有循环的分支跳转指令缩减为 1 条):

这种优化思想起源于计组 P2 使用一条 branch 指令实现循环所存在的 bug，如果仅在函数末尾增加一条 branch 分支，那么所形成的 do-while 循环不能避免一次也不执行的情况，因此应该在进入该循环前使用“一次性的判断器”判断是否执行，而后面部分使用 do-while 结构实现，通过这样的组合，使得 for 循环和 while 循环的分支跳转指令均变为 1 条，避免了使用 jump 指令。

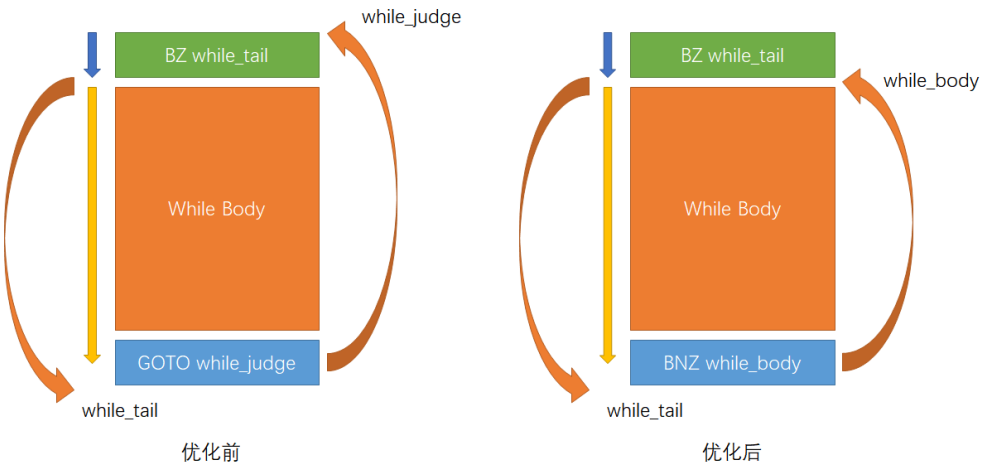

图 2 while 循环的分支跳转指令优化（引用自游子诺的讨论区帖子）

总结

编译课设不同于之前，更多给了我关于“设计”的直观感受，设计中间码与栈空间的规

范、设计类之间的关系、设计开发调试的支持工具……，在过程中体会到提早的设计对实践的高效性和可靠性的重要意义，当然目前所设计的结构尚存在许多改进之处，只有通过更多的练习才能提升，在今后的学习路上，我也需要更多关注该方面能力，打下坚实的基础。