编译原理实验 4 报告

姓名：黄奕诚　　**学号**：161220049

1　开发环境

- 操作系统：macOS Mojave 10.14.5
- 编辑器：Visual Studio Code 1.34.0
- C 编译器：Apple LLVM version 10.0.1 (clang-1001.0.46.4)
- Flex 版本：flex 2.5.35 Apple(flex-31)
- Bison 版本：bison (GNU Bison) 2.3

注：开发完成后已部署到 Ubuntu 18.04 上进行再测试

2　功能介绍

- 完成基本要求
- 在 SPIM 上通过了书中用例在内的多个测试

3　编译运行方法

- 编译：在 `Code/`目录下运行 `$ make` 即可，随后在项目根目录生成可执行文件 `parser`. **由于我在 macOS 上进行开发，将 `Makefile` 中编译选项`-lfl` 改为`-ll`，如果编译遇到问题不妨检查一下这里，也可以试一下先 `$ make clean` 再 `$ make`.**
- 运行：在项目根目录执行 `$ ./parser` [测试文件名] [目标代码输出文件] 即可. 然后在 SPIM 中导入文件即可.

4　实现重点细节

4.1　寄存器分配算法

采用了朴素寄存器分配算法，确保正确性优先. 因为每个运算操作都只能在寄存器之间进行，因此右值为变量时，从它的符号表中读取它存储的栈上的位置，`lw` 到寄存器中【见 `mips.c/make_m_op_m2r()`】. 左值为变量时，若它的符号表中尚未设置栈上的位置，则根据它的大小为它分配一个位置，否则读取该栈上位置，并把计算结果 `sw` 到内

存中【见 mips.c/make_m_op_get_m()】. 因为这个算法过于无脑，没有详细介绍的必要，便不展开.

4.2 内存管理

基本遵循了讲义中图 21 的函数活动记录结构. 因为我采用的是朴素寄存器算法，一个函数体中每个左值变量都需要一个对应的栈上的位置，对内存占用比较大. 我干脆直接把栈的大小设为很大的固定值，如果不是生成几千个局部变量基本搞不坏我的栈，不过放 Linux 上跑怕是也会爆栈. 变量在栈上的位置用了与 fp 寄存器的偏移量来指定，比起 sp 更加方便. 函数传参时直接依次放在了返回地址以上的位置.

4.3 目标代码生成

我的代码封装还是比较清晰的，以对 IR_CALL 的翻译来例:

```
case IR_CALL: { // x = call f
  MipsOperand *dst_label = make_m_op_func(code->u.sinop.op); // f
  make_mips_code_sp_add(-4); // sp = sp - 4
  make_mips_code_sw(ra_reg, make_m_op_arg_mem(0, sp_reg)); // sw ra 0(sp)
  make_mips_code_jal(dst_label); // jal f
  make_mips_code_lw(ra_reg, make_m_op_arg_mem(0, sp_reg)); // lw ra 0(sp)
  make_mips_code_sp_add(4); // sp = sp + 4
  MipsOperand *ret_reg = make_m_op_new_temp(); // reg1
  make_mips_code_move(ret_reg, v0_reg); // move reg1 v0
  MipsOperand *ret_mem = make_m_op_get_m(code->u.sinop.result, 4); // x
  make_mips_code_sw(ret_reg, ret_mem); // sw reg1 x
  return code->next;
}
```

生成目标代码的操作数函数名称统一用 make_m_op_ 开头，生成目标代码的函数名称统一用 make_mips_code_ 开头.

5 建议与吐槽

实验二、三、四的难度还是主要由实验一的词法、语法规则决定，我觉得这套实验语法规则偏简单，语义规则存在一些未定义行为（实验二的报告提过了），其它设计得比较正常. 实验三、四的自由度比较高，可以自由决定优化的程度，并且不对性能作出要求（如果规定要在 xx 条指令或者 xx 秒内跑完测试，难度就一下子上来了）