

编译原理Lab4实验报告

匡亚明学院

181240035 刘春旭

181240035@smail.nju.edu.cn

实验目标和结果

此实验的目标是把中间代码翻译成可以在MIPS上运行的目标代码。实现结果可以达到目标，采用最基础的朴素寄存器选择方法。

代码框架

1. 代码结构：

```
.  
├── common.h // 包含所有结构体的定义和必要的函数声明  
├── main.c // main函数  
├── lexical.l // Lab1 的语法检查  
├── syntax.y // Lab1的语法检查  
├── hash.c // Lab2的语义检查  
├── ir.c // Lab3的主体，负责翻译中间代码  
├── struct_syntable.c // Lab3的辅助代码，会在下面进行说明  
└── mips.c // [新增] Lab4的翻译代码部分
```

2. 指令选择：

基本按照讲义中的指示进行翻译。首先遍历一遍中间代码，记录必要信息，分配好变量空间，再按顺序扫描一遍中间代码即可产出目标代码。

3. 寄存器分配：

采用最简单的朴素寄存器分配算法。把所有的变量和临时变量 (t, v) 都提前在 `.data` 区分配好内存。用到的寄存器只有 `t0, t1, t2`，由于用完后立刻存回内存，所以每次使用寄存器时直接覆盖即可；并且也不会涉及到其他寄存器，比如 `a0~a3`，也避免了对它们的维护。

4. 栈管理:

栈中需要保存的部分只有传入的参数、该函数的所有变量和返回地址 `$ra`，函数调用的步骤是：

0. push所有变量到栈上
(1. 翻译到IR_ARG类型的中间代码，依次push参数到栈上)
2. push返回地址`$ra`到栈上
3. 调用函数 (`jal + move`) (--> 进入函数后依次从栈上恢复ARG的值，溢出到PARAM对应的变量中)
4. 恢复返回地址
(5. 恢复保存的参数，溢出到内存)
6. pop所有变量并溢出

--> 这里确实是一开始疏忽了，后来才意识到这种做法需要在每次CALL函数之前都push所有变量再pop所有变量，但是修改起来工程比较大，所以就暂时先这样了。这属于设计上的失误，其实完全可以在静态区分配空间，转而利用好栈上函数活动记录的临时变量区，这样可以减少很多代码量。

5. 关于数组:

对数组的空间分配并不是在 `.data` 中直接分配对应的数组大小，而是只为数组分配一个 `.word` 大小的空间（4Bytes）用于储存数组的指针；数组真正保存的位置在栈上：当遇到IR_DEC类型的中间代码时，做两件事：

- 栈指针减少对应数组大小 `addi $sp, $sp, -(ARRAY_SIZE)`
- 将栈指针保存到对应变量的位置上 `sw $sp, ARRAY`

这两件事的顺序不可以颠倒，因为数组的寻址过程是在基地址上加偏移量，但是栈增长的方向是负方向，所以栈顶对应的地址应为第一个数组元素的地址。这样一来，在翻译取地址操作时，直接使用 `lw` 命令就可以将地址加载到寄存器中。

6. 经过试验发现的关于函数的一个需要注意的地方：模拟器中函数名不能和指令名冲突，比如不能叫add、mul等，所以在每个函数之前都加上了一个个性化前缀。