编译原理lab4实验报告

姓名：张玄逸	学号：201220194
日期：2022.12.12	邮箱：1822771416@qq.com

实验内容

- 实现了必做内容（无选做）

测试方法

代码可使用Makefile进行编译，得到可执行文件parser。

在文件根目录下输入 ./parser [文件名1] [文件名2]即可执行，如 ./parser Test/test1.cmm out.s，前一个作为输入文件，后一个作为输出文件，也可以忽略[文件名2]，如./parser Test/test1.cmm，此时输出到控制台。

实验思路

- **总体**

  遍历lab3生成的中间代码，逐句翻译。

- **寄存器设置**

  首先，寄存器有一个名字，还需要存储里面的变量。为了进行替换，需要一个age变量。

```
typedef struct reg_descriper
{
    char name[6];
    struct Operand_ *var_store;
    int age;     // 已经保存的时间
} Reg_descriper; // 寄存器，保存里面存储的变量
```

  在初始化时按0-31号设置好寄存器的名字与可用寄存器。

```
strcpy(reg[29].name, "$sp"); // ESP
strcpy(reg[30].name, "$fp"); // EBP
strcpy(reg[31].name, "$ra");
// 可用寄存器：t0-t9,s0-s7，即8-25号
```

- **寻找寄存器（getreg）**

  该函数返回变量所在的寄存器，若没有则找到一个。首先应判断变量是否已入栈，若没有则需要将其入栈，为它分配存储空间（常数与*x类不用），并用它的offset成员记录自己存储的相对位置。因为所有变量都需要存在栈中，所以只需要记录与fp的相对位置就可以寻找到。比如offset = -8，则它在$fp - 8指向的位置。

```
void push_op(Operand *x)
{
    fp_offset -= 4;         // 数组的空间在DEC语句里处理
    x->offset = fp_offset; // 记录变量的位置
}
```

然后要更新寄存器的寿命，也就是把所有非空的寄存器age加一。然后先查找变量是否已经在寄存器中，在的话直接返回，否则尝试找空的寄存器。

都不行的话，需要根据age来选出最早被使用的寄存器来替换，还要将它里面的变量存回栈（通过offset找到位置）。

```
// 返回应该被替换的编号
target = oldest;
Operand *now = reg[oldest].var_store;
if (now->kind == PARA_operand || now->kind == TMP_operand)
{
    printf("  sw %s, %d($fp)\n", reg[oldest].name, now->offset);
}// 把里面的存回栈，注意只有参数和变量需要
```

接下来的加载有四种情况：常数使用 li 指令，普通变量用 lw ， &x 类要存入 x 的 offset 与 $fp 的和（也就是x的绝对地址），*x 类只需要返回 x 的寄存器即可，后续会在中间代码处理中完善。

- **函数进入与退出**

这一部分参考了PA的框架，仿照esp与ebp的变化完成了$fp与$sp的处理。以下就是每次进入与退出函数时的处理。

```
void into_func()
{
    fp_offset = 0;   // 重置偏移量
    var_list = NULL; // 变量为空
    // push ebp
    printf("  addi $sp, $sp, -8\n");
    printf("  sw $ra, 4($sp)\n");
    printf("  sw $fp, 0($sp)\n");
    // ebp=esp
    printf("  move $fp, $sp\n");
}
void leave_func()
{
    save_all_reg();               // 一个基本块的结束
    printf("  move $sp, $fp\n");  // esp=ebp
    printf("  lw $fp, 0($sp)\n"); // 恢复ebp旧值
    printf("  lw $ra, 4($sp)\n"); // 恢复ra旧值
    printf("  addi $sp, $sp, 8\n");
    printf("  jr $ra\n");
}
```

接下来是调用时的ARG与进入后的PARAM。

每个ARG语句都是一次变量的push，因此只需要每次存入并把栈顶指针 $sp 减四即可。

而PARAM随FUNCTION一同处理，因为压入参数时是逆序，因此取出应该从低到高地址，参数最低位置与函数内部变量最高位置相差8，因为有$ra和$fp。

- **基本块的判定与处理**

  由定义可推出，在label，return，goto，if，call 指令处基本块结束。此时调用save_all_reg，将寄存器中所有变量（除常量）存回栈中，并清空寄存器的状态。

总结与反思

- 无法获取输入

  因为虚拟机卡所以我在windows上跑qtspim，结果发现read函数无法获取键盘的输入。查资料得知，qtspim在windows下不识别微软输入法的英文。必须将键盘调为ENG键盘才可以进行有效输入。

- 循环时寄存器的存储

  在save_all_reg函数的调用上出现了错误，在翻译 IF 语句时需要给比较的两个变量找寄存器，我之前先保存了寄存器再给xy找寄存器，而这样导致刚被清空的寄存器又存进了两个变量，在跳转之后对寄存器的使用和存储造成了影响。正确顺序如下。

```
else if (p->kind == IF_in)
{
    int reg1 = get_reg(p->recop.op1);
    int reg2 = get_reg(p->recop.op2);
    save_all_reg(); // 一个基本块的结束
    ......// 注意：保存之后不能有任何对寄存器的操作！！！
```

- 参数顺序

  之前参数在获取时没有明确顺序导致弄反。实际上因为参数反向压入所以顺序与正常的相反，只需要依次往上取即可。

- 左右值

  之前并没有区别赋值语句的左右值导致lab4处理起来比较麻烦，因为有 *x = y 一类的语句所以不能用

  z = *x 来指代 *x，但这样在传参时就会出现ARG *x，write *x 类语句，增大了处理难度。所以修改lab3逻辑，左值保留*x，右值直接增加 z = *x，这样简化了lab4的处理。

- age更新

  一开始为变量找寄存器时，若它已在寄存器中则直接返回，忘记将该寄存器的age重置为0。这样就会导致意外的替换，比如 z = x + y，代码如下。

```
int reg3 = get_reg(result);
int reg1 = get_reg(op1);
int reg2 = get_reg(op2);
printf("  add %s, %s, %s\n", reg[reg3].name,
       reg[reg1].name, reg[reg2].name);
```

  注意到两个需要取值的操作数op1和op2（x和y）连续调用getreg，如果没有及时重置age，那么调用getreg(y)时可能就会替换掉age最大的reg(x)，导致里面的值发生变化，从而计算出错。