实验四：目标代码生成

2019.12.28

1 小组信息

组号	学号	姓名	邮箱
1	161220022	陈哲霏	novelist.chan@gmail.com
	161220024	成威	weicheng.nju@qq.com

2 实验说明

2.1 实验要求

- 对于满足给定8个假设条件的C--代码，将其中间代码转换成对应的目标代码(MIPS32汇编代码)

2.2 编译运行

- 在Lab/Code目录下执行make完成编译
- 执行./parser test1.cmm out1.s， 可以将test1.cmm的翻译结果输出到out1.s文件

3 实验内容

3.1 数据结构设计

- 寄存器描述符

```
struct RegDescription_ {
    char* name; // name of the reg
    bool ifFree; // if is free
    int dirty; // next use
    VarDescription var; // the variable stored
};
```

- 变量描述符，其中设计了一个大小为3的地址描述符数组，用以表示当前变量存在于寄存器、栈(内存)或是段中

```
union AddressDescription_{
    int regNo;
    int offset; // offset of stack
    char *name;  // name of variable in segment
};

struct VarDescription_ {
    char* varName;
    AddressDescription addrDescription[3]; // 0: REG; 1:
STACK; 2: SEGMENT

    VarDescription next;
};
```

3.2 函数设计

- 函数printObjectCode()中实现了中间代码到目标代码的转换输出，根据每一条中间代码中的操作类型来判断当前的目标代码输出成什么形式，对应为switch语句中的各个case

```
void printObjectCode(char *fileName) {
    // ...
    InterCode p = head->next;
    while(p != head){
        switch(p->kind){
            case LABEL: case FUNCTION:{
                char* op = getOperand(p->u.one.op);
                fprintf(fp, "%s:\n", op);
                break;
            }
            // ...
        }
    }
}
```

- getReg()函数实现了朴素寄存器分配算法，将所给操作数分配到寄存器上
- spillReg()函数实现了在寄存器分配满的情况下变量的溢出操作，将溢出变量存至栈上

3.3 寄存器使用情况

- 在寄存器分配算法中使用到的寄存器包括t0~t9和s0~s7共18个
- 在CALL、RETURN时参数使用a0~a4寄存器，返回值使用v0~v1寄存器
- 栈寄存器使用了sp和fp两个寄存器，分别保存栈顶与栈底指针。另外声明了全局变量NowOffset，用于在寄存器变量溢出时，对应地设置该变量在栈中的偏移量

4 实验总结

- 本次实验中遇到了较多困难，比如getReg()函数实现时原本想实现局部寄存器分配算法，但由于未分基本块，无法实现算法内部的部分逻辑，故采用了朴素的寄存器分配算法
- 在考虑使用sp还是fp来处理栈指针时遇到问题，我们最终讨论后决定使用两者，并添加一个全局变量来记录栈偏移量
- 在完成编译器的同时收获了快乐