编译原理 Lab4 实验报告

211220028 党任飞

一、实现功能

实现了将符合规范的c--源文件转化为MIPS格式机器指令的部分编译器。其中词法分析、语法分析、语义分析、中间代码生成与之前的一致，没有修改。在此基础之上，增加 `objCode.c` 和 `objCode.h` 两个文件，实现 `generateObjectCode()` 接口并在main函数中调用，以实现从IR到目标指令的翻译。

二、程序运行方式

执行如下指令即可：

```
> make
> ./parser test.cmm out.s
```

三、核心思路

从IR到机器指令的翻译核心函数为 `generateObjectCode()`，其基本结构如下：

```
void generateObjectCode(FILE* file)
{
    initRegisters();
    InterCodes temp = interCodeHead->next;
    while(temp != interCodeHead){
        switch(temp->code->kind){
            case LABEL_IC:
            {
                genLabelCode(temp, file);
                break;
            }
            ......
        }
        temp = temp->next;
    }
}
```

可见，本函数实现的是几乎一对一的翻译过程，没有考虑滑动窗口组合优化之类的技术。每一个InterCode就对应一个 `genXXXCode()` 函数。

其中最重要的是 `genFunctionCode()` 函数。由于本实验的C--语法规定了没有全局变量，因此其实所有变量都只需要分配到栈上，这一过程在上述函数中完成的，基本结构如下：

```
void genFunctionCode(InterCodes temp, FILE* file)
{
    // preliminarily create vars and allocate spaces in stack here
    ......

    InterCodes tempIC = temp->next;
    while(tempIC->code->kind == PARAM_IC){
        // allocate PARAM memory
        ......
    }
    // deal with other InterCodes until meet another FUNCTION
    while(tempIC!=interCodeHead && tempIC->code->kind!=FUNCTION_IC){
        switch (tempIC->code->kind)
        {
            case GET_ADDR_IC:
            case GET_CONTENT_IC:
            case WRITE_ADDR_IC:
            case ASSIGN_IC:
            {
                createVarDesc(tempIC->code->u.binOP.op1);
                createVarDesc(tempIC->code->u.binOP.op2);
                break;
            }
            ......
        }
        tempIC = tempIC->next;
    }
    resetRegisters();
}
```

此外，本实验实现中只使用了t0-t7，s0-s7寄存器。并且使用朴素的分配方法，在所有SUB、ADD、MUL、DIV、CALL等写值的指令最后都把值写回栈中。即便如此，如果真的出现所有寄存器全部用完的情况，使用类似时钟CLOCK的方法挑选出一个被写回内存。