编译原理实验四：目标代码生成

组长 161130118 尹浚宇
组员 171850524 刘扬

一、实现功能

在词法分析、语法分析、语义分析和中间代码生成程序的基础上，通过指令选择、寄存器选择以及栈管理，将 C--源代码翻译为包含伪指令的 MIPS32 指令序列。

二、实现方法

1. 指令选择

由于实验 3 中我们使用的是线性中间代码，所以这里我们可以简单地通过逐条遍历代码并结合翻译表来给出目标代码来完成指令选择的工作. 由于实验 3 中对中间代码已经进行了大量优化，所以这里并不需要再进一步优化目标代码，这样反而会生成一些需要将优化还原才能得到正确目标代码的情况. 下面是我们采用的翻译表，这是教程里给出的.

中间代码	MIPS32指令
LABEL x:	x:
x := #k	li reg(x)[1], k
x := y	move reg(x), reg(y)
x := y + #k	addi reg(x), reg(y), k
x := y + z	add reg(x), reg(y), reg(z)
x := y - #k	addi reg(x), reg(y), -k
x := y - z	sub reg(x), reg(y), reg(z)
x := y * z[2]	mul reg(x), reg(y), reg(z)
x := y / z	div reg(y), reg(z) mflo reg(x)
x := *y	lw reg(x), 0(reg(y))
*x = y	sw reg(y), 0(reg(x))
GOTO x	j x
x := CALL f	jal f move reg(x), $v0
RETURN x	move $v0, reg(x) jr $ra
IF x == y GOTO z	beq reg(x), reg(y), z
IF x != y GOTO z	bne reg(x), reg(y), z
IF x > y GOTO z	bgt reg(x), reg(y), z
IF x < y GOTO z	blt reg(x), reg(y), z
IF x >= y GOTO z	bge reg(x), reg(y), z
IF x <= y GOTO z	ble reg(x), reg(y), z

2. 寄存器选择

考虑到程序中各种嵌套循环，过程调用，递归等复杂情况，我们这里选用的是实现难度最小的朴素寄存器分配算法. 这里我们将 7 号至 25 号寄存器均作为通用寄存器使用，将 5 号和 6 号寄存器作为 IFGOTO 语句专用的寄存器.

在程序中需要考虑分配寄存器的地方，我们的代码先考虑有没有空闲寄存器，如果有则直接选择空闲寄存器，如果没有则从已分配的寄存器中选择一个，并将其内容溢出回内存.

特别地，在处理一些可能改变寄存器内值的指令生成时，我们选择每次执行完之后都将该寄存器里的内容溢出回内存，这也正是书上介绍的朴素算法的基本思想.

3. 栈管理

我们的栈管理方案基本按照 i386 的栈管理方案来实现. 具体而言，我们对于栈上所有变量的寻址都是基于帧指针(这里为 fp)的偏移来实现的. 这就要求我们维护每个变量相对于当前帧指针的偏移值. 经过考虑后，我们选择使用下面的数据结构来表示变量.

```
struct Data_ {
    char* name;
    int offset;
    struct Data_* next;
};
```

其中 name 是变量名, offset 是相对于帧指针的偏移量, next 指针是为开地址哈希表准备的.

考虑到效率问题，这里查找和存储所有变量的数据结构我们选用的是哈希表，其哈希函数为实验 2 教程中给出的哈希函数.

由于模仿 i386 的栈管理方式，在过程调用时，我们的代码在栈上按顺序逐个压入参数，之后压入返回地址和 old fp，并将新的 fp 指向当前的栈顶. 在过程返回时，按照调用时压入参数的相反顺序恢复栈即可.

三、编译方式

使用自带的 makefile 进行编译