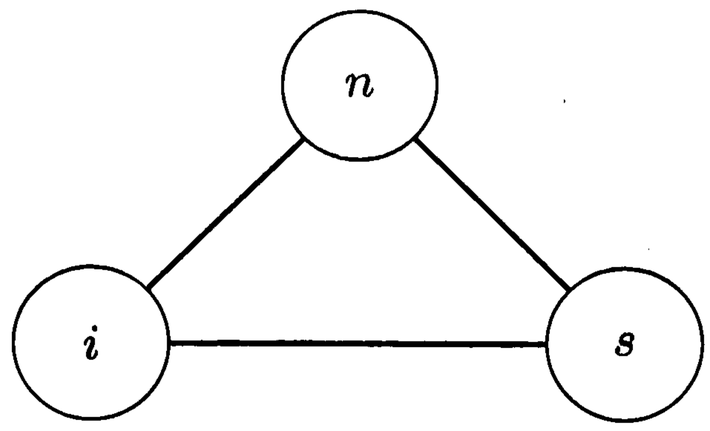

图 1.4　程序的干涉图

1.4　寄存器分配

程序中包括声明变量以及编译器自动生成的临时变量（以下统称为变量），为了让程序在目标机器上执行，编译器必须给这些变量分配合理的存储。编译器可以采用一个非常保守的策略，即把所有变量都存储在内存中，仅当需要进行计算时，才将这些变量临时加载到物理寄存器中，计算完成后把结果重写存回内存，这种策略称为栈分配策略。尽管栈分配策略非常简单，并且易于实现，但由于在目前的计算机体系结构中，访问寄存器的速度远比访问内存快，因此，为了尽可能提高目标程序的运行效率，编译器应该尽可能将这些变量优先分配到寄存器中，仅当物理寄存器不够用时，才将变量存放在内存中，这个过程称为寄存器分配（register allocation，RA）。从这个角度看，寄存器分配是编译器后端进行的一种优化。

寄存器分配是一个非常重要的问题，并且可能是所有编译优化中最重要的一个，引用 Hennessy 和 Patterson 的说法：

> 寄存器分配由于在代码加速以及其他优化方面扮演核心角色，从而成为重要的——即便不是最重要的——编译器优化。

因此，任何一个用于实际生产的编译器，都必须精心地选择和实现高效的寄存器分配算法。

1.4.1　栈分配策略

尽管编译器用栈分配策略编译生成的目标代码效率不高，但栈分配策略仍然非常重要，主要是因为：第一，由于栈分配比较简单，编译器后端可同时实现栈分配和寄存器分配，并将栈分配作为后端实现的基线，用来对寄存器分配进行结果对比

验证测试；第二，如果需要对编译器生成的可执行文件进行调试，则对变量进行栈分配会更加方便（如设置断点、设置观察点等调试操作），等等。

栈分配策略主要分成两个阶段：

(1) 变量分配：对给定程序 P 中的每个变量 x 都分配一个栈帧上的偏移 l_x，在程序执行期间，该位置 l_x 始终被变量 x 独占。

(2) 程序改写：假定我们考虑的机器模型是一个精简指令集体系结构（Reduced Instruction Set Computer，RISC），则所有的计算必须都在物理计算器中进行，因此，我们需要加入额外的数据存储指令，并对原有指令进行改写。

第一个阶段的实现比较简单，我们接下来重点讨论第二个阶段。第二个阶段可基于一个语法制导的局部改写规则实现，这个规则基于对程序各种可能语法形式的归纳。假设目标机器上一共有 K 个物理寄存器 $r_1,\cdots,r_K$，则我们可给出如下的两个函数：

$$\mathcal{S}(s)=\cdots$$
$$\mathcal{J}(j)=\cdots$$

分别对语句 s 和控制转移 j 进行改写。

基于对语句 s 语法形式的归纳，我们可给出如下的程序改写规则：

$$\begin{aligned}\mathcal{S}(y=\tau(x_1,\cdots,x_n)) &= r_1=r_{BP}[l\_x_1];\\ &\cdots;\\ &r_n=r_{BP}[l\_x_n];\\ &r_1=\tau(r_1,\cdots,r_n);\\ &r_{BP}[l\_y]=r_1;\\ \mathcal{S}(y=f(x_1,\cdots,x_n)) &= r_1=r_{BP}[l\_x_1];\\ &\cdots;\\ &r_n=r_{BP}[l\_x_n];\\ &r_1=f(r_1,\cdots,r_n);\\ &r_{BP}[l\_y]=r_1;\\ \mathcal{S}([y]=x) &= r_1=r_{BP}[l\_x];\\ &r_2=r_{BP}[l\_y];\\ &[r_2]=r_1;\\ \mathcal{S}(y=[x]) &= r_1=r_{BP}[l\_x];\\ &r_1=[r_1];\end{aligned}$$

$$r_{BP}[l\_y] = r_1;$$
$$S(y = x) = r_1 = r_{BP}[l\_x];$$
$$r_{BP}[l\_y] = r_1;$$

其中的寄存器 r_{BP} 表示函数栈帧的基址寄存器，所有待分配的变量 x 的偏移 l_x 都相对该基址。以算术运算语句 $y = \tau(x_1, \cdots, x_n)$ 为例，首先，编译器将 n 个运算数 $x_1, \cdots, x_n$ 分别读入 n 个寄存器 $r_1, \cdots, r_n$ 中，然后再执行算术运算 τ 并将运算结果存入寄存器 r_1 中（复用了该寄存器），最后将寄存器 r_1 中的结果写回内存中变量 y 对应的偏移地址 l_y 处。一般地，算术运算只有 2～3 个操作数，因此也最多用到 2～3 个物理寄存器。

其他语句的规则类似，我们留给读者逐个分析。唯一需要强调的是对函数调用 $y = f(x_1, \cdots, x_n)$ 的转换规则，如果在具体的指令集体系结构上，还需要考虑具体的调用规范等细节，此处做了简化。

对控制转移语句 j 的转换规则 $\mathcal{J}()$ 定义如下：

$$\mathcal{J}(goto\ L) = goto\ L;$$
$$\mathcal{J}(if(x, cond, y, L1, L2)) = r_1 = r_{BP}[l\_x];$$
$$r_2 = r_{BP}[l\_y];$$
$$if(r_1, cond, r_2, L1, L2);$$
$$\mathcal{J}(return\ x) = r_1 = r_{BP}[l\_x];$$
$$return\ r_1;$$

和函数调用的规则类似，函数返回的规则 $return\ x$ 同样略去了与具体指令集体系结构相关的调用规范，除此之外，这些规则都比较直接，具体的分析过程作为练习留给读者。

总结下来，无论是从理论上还是从实现上来看，基于栈分配的寄存器分配都比较简单且容易实现，编译器实现者都应该考虑先实现这个版本，得到一个也许并不能高效生成代码，但运行结果正确的程序，并以此为基线实现更复杂寄存器优化算法。在任何情况下，简单且正确的代码，总好过复杂且易错的代码。

1.4.2　寄存器分配策略

函数粒度的寄存器分配一般被称为全局分配，全局寄存器分配仍然是一个困难问题，理论上已经证明：一般的全局分配问题的难度是 NP 完全的，这意味着目