第 1 章

控制流图

控制流图是一种基于有向图的编译器中间表示，在程序分析和编译优化中起着重要作用。在本章，我们将首先介绍控制流图的定义、构建方法及其基本操作；接着，我们将讨论基于控制流图定义的不同粒度的程序分析和编译优化技术，为后续章节深入探讨相关技术奠定基础。

1.1 控制流图的定义

编译器对程序进行分析或者优化，需要基于对程序的合适的中间表示进行。**控制流图**（Control-flow Graph，CFG）是一种基于有向图的程序中间表示形式，可记为 $G = (V, E)$，其中有向图 G 中的节点集合 V 由控制流图的**基本块**（Basic Block, BB）组成，每个基本块包含一段连续的语句序列；有向边集合 E 中的任意一条有向边 $(V_f, V_t) \in E$ 刻画了流图 G 中的节点 V_f 到 V_t 之间的跳转关系。

我们可以用如下扩展的上下文无关文法，来给出控制流图的形式定义。文法中的符号“*”是 Kleene 闭包，表示该语法实体出现 0 次或任意多次；例如，$S*$ 代表语句 S 出现 0 次或

任意多次。

$$
\begin{aligned}
E &::= n \mid x \mid x_1 \oplus x_2 \mid f(x) \\
S &::= x = E \\
J &::= \texttt{if}(x; L_1; L_2) \mid \texttt{jmp}\ L \mid \texttt{ret}\ x \\
B &::= L\ S*\ J \\
F &::= f(x)\{B*\} \\
P &::= F*
\end{aligned}
$$

整个程序 P 由若干个函数 F 组成，每个函数 F 包含函数名 f、函数参数 x 以及函数体 $B*$。其中，函数体由零个或任意多个基本块 B 组成。为简单起见，我们假定函数只有一个参数 x，并默认其为整数类型，这并不影响对控制流图的一般讨论。在实际编译器实现中，可以根据实际需要对该文法进行拓展，但这更多的是增加工程上的工作量，而不增加理论上的难度。

每个基本块 B 包括一个标号 L 作为该基本块的唯一标识，后跟若干条语句 S，最后以一条控制转移语句 J 结尾。

语句 S 只包含对变量的赋值 $x = E$ 这一种语法形式，它将右侧表达式 E 的计算结果赋值给左侧的变量 x。实际的编译器实现一般会包含多种不同的语句形式，例如对内存地址的读写等，可以根据实际需要对该文法进行拓展。

表达式 E 包括几种不同的语法形式。表达式 n 表示立即数 n；表达式 x 表示变量；表达式 $x_1 \oplus x_2$ 表示对变量 x_1 和 x_2 的二元运算，其使用了抽象运算符号 $\oplus$ 指代任意的二元运算，如加、减、乘、除等；最后，函数调用表达式 $f(x)$ 使用参数 x 调用函数 f。

控制转移语句 J 包含三种形式。条件转移语句 `if` 根据变量 x 的值，来决定跳转目标，当 x 为真（即 $x \neq 0$）时，控制流跳转到标号 L_1 代表的内存地址处；反之，如果 x 为假（即 $x = 0$），则控制流跳转到标号 L_2 代表的内存地址处；无条件转移语句 `jmp` 直接跳转到标号 L 处；函数返回语句 `ret` 退出当前函数的执行，并将返回值 x 返回给函数调用者。

对上述给定的程序控制流图的定义，有几个关键点需要注意。首先，函数 F 中包含唯一的入口基本块 B，函数 F 从该基本块 B 开始执行；函数 F 中可以包含一个或多个退出块，函数在这些退出块结束执行。

其次，每个基本块 B 都包含一段最大连续的顺序语句序列，控制流只能从基本块 B 的第一条语句进入该块，顺序执行基本块中的每条语句后，在最后一条跳转语句处进行跳转，从而退出该基本块。语句的执行既不能从基本块中间的某条语句开始，也不能从基本块的中间退出执行。

为了更直观的说明控制流图的表示，考虑如下求和函数的 C 语言代码示例：

```c
int sum(int n){
    int s = 0, i = 0;
    while(i <= n){
        s += i;
        i += 1;
    }
    return s;
}
```

该函数编译得到的控制流图代码如下所示。控制流图代码由 L_0 到 L_3 四个基本块组成，其中基本块 L_0 和 L_2 由无条件跳转 `jmp` 结尾，基本块 L_1 由有条件跳转 `if` 结尾，而基本

块 L_3 由返回语句 `ret` 结尾。

```
int sum(int n){
   L0: s = 0;
       i = 0;
       jmp L1;
   L1: t = i <= n;
       if(t, L2, L3);
   L2: s = s + i;
       i = i + 1;
       jmp L1;
   L3: ret s;
}
```

为了进一步直观表示程序的控制流图，我们经常将控制流图画成显式的有向图形式；其中，控制流图的基本块作为有向图的节点，而控制流图的边作为有向图的边。例如，对上述的程序示例，其控制流图的显式有向图表示如图 1.1所示。

1.2 控制流图的构建

编译器需要基于程序的其它初始表示形式，来构建程序的控制流图。而根据程序的初始表示形式相对于控制流图的抽象层次的高低，控制流图的构建算法可分为两类：自顶向下的构建方法和自底向上的构建方法。在自顶向下的构建方法中，编译器从比控制流图更高层的程序表示出发，向下构建程序控制流图；在许多编译器实现中，高层表示一般是程序的抽象语法树等表示形式。而在自底向上的构建方式中，编译器从比控制流图更底层的程序表示出发，向上构建程序的控制流图；在许多编译器实现中，底层表示一般是程序的汇编甚至是二进制代码。

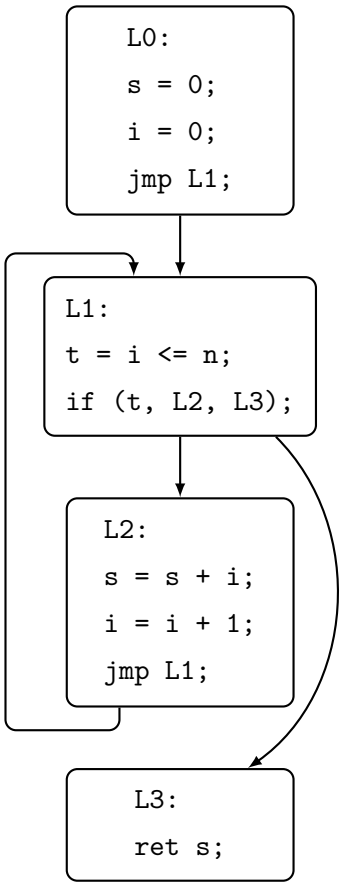


图 1.1: sum 函数的控制流图

1.2.1 源语言定义

为讨论控制流图的自顶向下构建方法，我们先给出一个简单的源语言定义，并从该源语言出发，讨论控制流构建算法。该源语言的抽象语法，由如下扩展的上下文无关文法给出。尽管该语言相对简单，但其包括了常见的代表性程序结构。

$$E ::= n \mid x \mid E \oplus E \mid f(E)$$

$$S ::= x = E \mid \texttt{if}(E; S; S) \mid \texttt{while}(E; S) \mid \texttt{return}\ E \mid S*$$

$$F ::= f(x)\,\{S\}$$

$$P ::= F*$$

语法中程序 P、函数 F 的定义类似于第1.1节给出的控制流图中的相应定义，此处不再赘述。

语句 S 有五种不同的语法形式。赋值语句 $x = E$ 将表达式 E 赋值给变量 x；`if` 语句根据表达式 E 的值，决定执行第一个分支还是第二个分支；`while` 语句根据表达式 E 的结果，来执行循环体中的语句 S 或直接退出循环；`return` 语句退出函数执行，并返回表达式 E 作为返回值；语句块 $S*$ 是由零个或多个语句构成的一个语句序列。

表示式 E 包括四种语法形式。和控制流图（第1.1节）中给出的表达式定义不同，这里的表达式包含显式的递归结构；例如，二元运算 $E_1 \oplus E_2$ 的左右运算数 E_1 和 E_2 分别都是表达式。

1.2.2 自顶向下构建

自顶向下的构建方法通常从程序的高层表示（如抽象语法树等），来构建控制流图。而由于程序高层表示一般包括结构化控制流（如条件判断、循环等）定义、以及递归形式的表达式定义，因此，自顶向下的构建往往使用递归算法。

算法1.1给出了过程 `transExp`，它将抽象语法树的表达式 E，转换为控制流图相应语法结构。`transExp` 接受一个表达式 e 作为参数，并返回包括两个字段 $< x, S >$ 的二元组作为结果；其中变量 x 是表达式 e 翻译得到的结果变量，而 S 是翻译过程中生成的控制流图中的语句序列。

翻译函数 `transExp` 采用了语法制导的方法，根据表达式 e 的语法类型，应用不同的翻译规则。对于常量表达式 n，函数将调用辅助函数 `freshVar`，新生成一个全新的变量 y，并

算法 1.1 表达式 e 的翻译算法

输入： 表达式节点 e

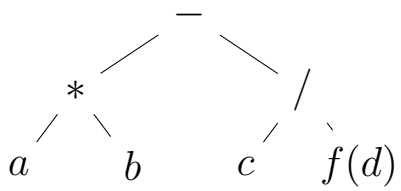
输出： 一个二元组 <x, S>，其中 x 是表达式 e 翻译得到的结果变量，S 是转换过程中生成的控制流图的语句序列

```
function transExp(e)
    switch e do
        case n:
            y = freshVar()
            return <y, {y = n}>
        case x:
            return <x, {}>
        case e1 ⊕ e2:
            <x1, S1> = transExp(e1)
            <x2, S2> = transExp(e2)
            y = freshVar()
            return <y, S1@S2@{y = x1 ⊕ x2}>
        case f(e):
            <x, S> = transExp(e)
            y = freshVar()
            return <y, S@{y = f(x)}>
```

将变量 y 以及赋值语句 $y = n$ 组成的二元组 $< y, \{y = n\} >$ 作为返回值返回。对变量 x，由于它已经是原子表达式，因此无需进行翻译，直接返回它本身以及一个空的语句序列组成的二元组 $< x, \{\} >$ 即可。对于二元运算表达式 $E_1 \oplus E_2$，算法首先分别对两个表达式 E_1 和 E_2 进行递归翻译，并分别返回两个二元组 $< x_1, S_1 >$ 和 $< x_2, S_2 >$ 作为中间结果，然后，算法将返回一个新的二元组 $< y, S_1@S_2@\{y = x_1 \oplus x_2\} >$ 作为翻译结果；其中变量 y 是生成的全新的赋值变量作为结果，而符号 @ 表示语句序列的拼接操作。对于函数调用表达式 $f(e)$，算法首先对参数表达式 e 进行递归调翻译，生成一个结果二元组 $< x, S >$，随后，算法将返回二元组 $< y, S@\{y = f(x)\} >$

作为最终结果。

作为示例，考虑表达式 $a * b - c/f(d)$，其抽象语法树如下所示：



算法 `transExp` 从根节点“$-$”开始递归向下处理，假设算法执行时依次为 $*$、$f(d)$、$/$ 和 $-$ 四个节点生成的全新变量分别为 y_1、y_2、y_3 和 y_4，则翻译后生成的语句序列如下所示：

```
y1 = a * b;
y2 = f(d);
y3 = c / y2;
y4 = y1 - y3;
```

基于对表达式 E 的控制流图构建算法1.1，算法1.2给出了对语句 S 的控制流图构建算法 `transStm`。算法使用两个全局变量 C 和 $blocks$，分别记录当前正在处理的基本块和已创建并加入到函数控制流图中的基本块。算法每当生成一条新语句 s，都会将其追加到基本块 C 的末尾，而每当新创建一个新的基本块 b，都会将其追加到 $blocks$ 末尾。

算法 `transStm` 同样采用语法制导的策略，对语句 S 语法形式进行分情况讨论，根据不同的节点类型应用不同的翻译规则。

对于赋值语句 $x = E$，算法 `transStm` 首先调用算法 `transExp`，将其中的表达式 E 翻译为控制流图表示，并将翻译得到的语句序列 T 以及语句 $x = y$ 分别添加到当前基本块 C 末尾。

算法 1.2 语句 S 的翻译算法

输入：语句 S 的抽象语法树表示

输出：对应语句的控制流图表示

```
1: C = null;
2: blocks = {};
3: function setCurrent(b)
4:     C = b
5:     blocks = blocks@{b}    // append b to blocks
6: function transStm(S)
7:     switch S do
8:         case x = E:
9:             <y, T> = transExp(E)
10:            C = C@T@{x = y} // append S and x = y to the basic block C
11:        case T*:
12:            for each statement U ∈ T* do
13:                transStm(U)
14:        case if(E, S1, S2):
15:            T = freshBlock(); E = freshBlock(); M = freshBlock()
16:            <y, S> = transExp(E)
17:            C = C@S@{if(y, T, E)}
18:            setCurrent(T);   transStm(S1);   C = C@{jmp M}
19:            setCurrent(E);   transStm(S2);   C = C@{jmp M}
20:            setCurrent(M)
21:        case while(E, S1):
22:            // H, B, M: the entry, loop body, and exit block, respectively
23:            H = freshBlock(); B = freshBlock(); M = freshBlock()
24:            C = C@{jmp H}
25:            setCurrent(H); <y, S> = transExp(E); C = C@S@{if(y, B, M)}
26:            setCurrent(B);   transStm(S1);   C = C@{jmp H}
27:            setCurrent(M)
28:        case return E:
29:            <y, T> = transExp(E)
30:            C = C@T@{ret y}
```

对于语句序列 $T*$，算法 `transStm` 依次递归翻译序列中

的每条语句 U。

对于条件语句 `if`(E, S_1, S_2)，算法首先创建 T、E 和 M 三个基本块，分别对应条件语句的真块、假块以及合并块。然后，算法递归调用函数 `transExp`，将条件表达式 E 翻译为控制流图，并根据条件表达式 E 的翻译结果 y，生成一条条件跳转语句 `if`(y, T, E)，并将生成的语句序列 S 和条件跳转语句都追加到基本块 C 的末尾。接下来，算法将基本块 T 设置为当前块，然后递归调用函数 `transStm`，翻译 `if` 语句的真分支 S_1，翻译完成后，在当前块 C 的末尾，生成一条直接跳转指令 `jmp` M 跳转到基本块 M。之后，算法使用类似的步骤，处理条件语句的假分支语句 S_2。最后，算法将基本块 M 设置为当前基本块 C，并进行后续语句的翻译。

对于循环语句 `while`(E, S_1)，算法首先创建循环入口 H、循环体 B 和循环出口 M 三个基本块。然后，算法生成一条跳转语句 `jmp` H 跳转到循环入口块 H。接着，算法将当前块 C 设置为 H，并递归翻译循环的条件判断表达式 E，接着在当前块 C 末尾生成一条条件跳转语句 `if`(y, B, M)，分别跳转到循环体 B 和循环出口块 M。接下来，算法对循环体语句 S_1 进行翻译，即将当前块 C 设置为 B，并递归调用函数 `transStm` 翻译循环体 S_1，处理完成后，生成一条直接跳转指令 `jmp` H，跳转到循环入口块 H。最后，算法将合并块 M 设置为当前块 C。

算法1.3中的函数 `transFunc` 翻译每个函数 $f(x)\,\{S\}$，并为其创建控制流图。函数 `transFunc` 首先为函数 f 创建一个入口基本块 b 并将其设置为当前块；接着，算法调用函数 `transStm` 翻译函数体语句 S。最后，函数返回构建得到的函数的控制流图表示 $f(x)\,\{blocks\}$。

算法 1.3 函数 F 的翻译算法

输入： 一个函数 F 的抽象语法树表示

输出： 对应函数的控制流图表示

```
1: function transFunc(F = f(x) {S})
2:     b = freshBlock(); setCurrent(b);
3:     transStm(S);
4:     return f(x) {blocks};
```

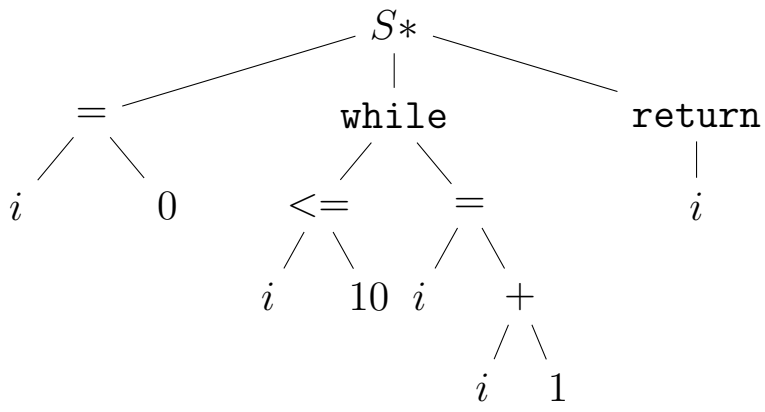
对上述控制流图的构建算法，有几个关键点需要注意。首先，需要特别注意算法1.2对新生成语句的添加位置。具体的，算法中新生成的语句都应该追加到当前基本块 C 后，而不能直接追加到新创建的基本块中。例如，在对条件语句 `if` 进行翻译时，在算法第 18行，不能将 `jmp` M 直接添加到基本块 T 中，虽然 T 表面上看起来是 `if` 的真块，但由于 `if` 语句 `then` 分支可能嵌套了其它语句，因此在递归调用 `transStm` 的过程中，算法可能重新设置了当前块 C，亦即当前块 C 未必是基本块 T。

其次，要特别注意将一个基本块追加到基本块序列 $blocks$ 的时机。算法只有在将某个基本块 B 设置当前块 C 时，才会将 B 追加到 $blocks$ 中，换句话说，只有之前所有的基本块都处理完，真正要为某个基本块 B 生成控制流图的语句时，才会将基本块 B 放入 $blocks$ 中，算法按这样的步骤执行，可以保证 $blocks$ 中的基本块尽可能满足伪拓扑有序（见第1.3.2小节）。尽管基本块排列的顺序并不会影响程序执行结果的正确性，但是一个良好的基本块排列可以提高指令的局部性，从而提升程序的执行效率。

最后，限于篇幅原因，算法1.2仅给出了部分高层控制流语句的控制流图构建算法。在实际中，对其它形式的高层控制流语句，只需对上述算法进行扩展即可。例如，对于 `for`

循环语句，可以参照 `while` 语句的处理给出对应的翻译规则。

作为示例，考虑如下的抽象语法树：



其根节点所代表的语句序列 $S*$，从左到右依次包含一个赋值语句、循环语句和返回语句。算法1.2从左到右依次对语句序列中的每条语句进行处理（即按第二个分支进行处理）。首先，算法将赋值语句 $i = 0$ 生成到入口块 B_0 中。接着，算法为 `while` 语句生成三个基本块 B_1、B_2 和 B_3（即入口块、循环体块、和出口块），并分别将条件判断 $i <= 10$ 和循环体内的语句 $i = i + 1$ 放置到基本块 B_1 和 B_2 中，并在每个块的末尾，分别追加合适的跳转语句。最后，算法为 `return` 语句，生成基本块 B_3。算法最终生成的控制流图如下所示：

```
B0: i = 0;
    jmp B1;
B1: t = i <= 10;
    if(t, B2, B3);
B2: i = i + 1;
    jmp B1;
B3: return i;
```

1.2.3 自底向上构建

自底向上的控制流图构建方式，接受比控制流图更低层的代码表示，向上构建并输出控制流图表示。一般的，自底向上的构建方式常用于构建非结构化代码的控制流图表示。例如，在反汇编场景中，反汇编器接受汇编指令序列作为输入，自底向上构建程序的控制流图表示用于后续分析。另外，即使是从高层代码出发构建控制流图，如果高层语言中包含 `goto` 等非结构化控制流语句，也可以先将高层代码编译为线性语句序列，然后再采用自底向上方式，构建其控制流图。

自底向上的控制流图构建，关键在于识别基本块的首指令或称首领 （leader），亦即基本块的第一条指令。回顾基本块的定义，每个基本块都是最大连续且不包含跳转的指令序列，因此，确定基本块首领的规则如下：

1. 每个函数的第一条指令是一个首领；
2. 无条件或有条件跳转指令的目标指令是一个首领。
3. 紧跟在无条件或有条件跳转之后的指令是一个首领。

根据上述规则，算法1.4给出基于首领思想的控制流图构建算法。函数 `findLeaders` 对输入的线性语句序列 S 进行一遍扫描，按照上述给出的三条规则，识别指令序列 S 中所有的首领，并记录在集合 $leaders$ 中。该函数最后将首领集合 $leaders$ 中的首领，按出现顺序排序并返回。

函数 `buildCfg` 接收一个函数的线性指令序列 $S[]$ 作为输入，构建并返回基本块序列 $blocks$ 作为输出，该基本块序列 $blocks$ 对应函数的控制流图。函数 `buildCfg` 首先调用函

算法 1.4 基于首领思想的自底向上控制流图构建算法

输入： 程序中一个函数的线性指令序列 $S[]$

输出： 一个基本块序列，对应该函数对应的控制流图表示

```
1:  function findLeaders(S[])
2:      leaders = {}
3:      leaders∪ = {S[0]}
4:      for each instruction s ∈ S do
5:          if s's predecessor is a jump then
6:              leaders ∪= s    // add s to leaders
7:          else if s is a an instruction jmp L then
8:              leaders ∪= {L}
9:          else if s is a an instruction if(x, L1, L2) then
10:             leaders ∪= {L1, L2}
11:     return sort(leaders)
12: function buildCfg(S[])
13:     blocks[] = {}
14:     leaders = findLeaders(S)
15:     for each leader l ∈ leaders do
16:         t = the next leader after l
17:         b = a block with the instruction sequence S[l..t − 1]
18:         blocks = blocks@{b}    // append b to the end of blocks
19:     insert jump instruction for the basic block in blocks lacking jump instruction
20:     return blocks
```

数 `findLeaders`，得到所有的首领。接着，函数 `buildCfg` 根据每个首领 l 以及它后继的首领 t，确定位于同一个基本块中的指令序列 $S[l..t-1]$，并使用该序列构建一个基本块 b，并将其添加到基本块列表 $blocks$ 中。此外，由于默认跳转的存在，前一个基本块可能默认跳转到其后继块，因此，为了满足基本块的语法结构约束，算法在创建好所有的基本块后，还需要在包含默认跳转的基本块末尾，插入必要的显式跳转指令。

作为示例，考虑下面左侧的求和程序 `sum` 的类汇编代码，其跳转语句的目标都是显式的伪代码地址。例如，第 5 行的 `if`

```
1  int sum(int n) {
2     s = 0;
3     i = 0;
4     t = i <= n;
5     if(t, 6, 9);
6     s = s + i;
7     i = i + 1;
8     jmp 4;
9     ret s;
10 }
```

```
1  int sum(int n) {
2  B0: s = 0;
3      i = 0;     jmp B1;
4  B1: t = i <= n;
5       if(t, B2, B3);
6  B2: s = s + i;
7      i = i + 1;
8      jmp B1;
9  B3: ret s;
10 }
```

语句根据判断条件 t，分别跳转到第 6 行（即循环体）或第 9 行（即循环退出）。对该程序应用上述算法1.4，函数 findLeaders 首先识别所有的首领，并返回首领集合 $leaders = \{2, 4, 6, 9\}$，这意味着首领分别位于程序的第 2、4、6 和 9 行。接下来，算法为每个首领确定位于同一个基本块中的指令序列，可得到每个首领的基本块所包含的指令区间分别为 $[2, 3]$、$[4, 5]$、$[6, 8]$ 以及 $[9, 9]$。接着，算法使用这四个指令区间分别构建一个对应的基本块 B_0 到 B_3，每个基本块包含该区间内的指令。最后，由于基本块 $B_0 = [2, 3]$ 最后一条指令 $i = 0$ 是默认跳转指令，为了满足基本块的语法形式约束，算法在其最后插入一条无条件跳转语句 jmp B_1，跳转到其后继基本块 B_1。算法运行结束后，得到的结果控制流图如右图所示。

1.3 控制流图操作

编译器在对控制流图进行分析和优化的过程中，需要对控制流图完成各种操作和变换。而控制流图的结构本质上是一个有向图，因此对有向图的操作都适用于控制流图。本节将讨论图的遍历、节点排序、关键边切分、以及最大强连分量

算法 1.5 广度优先遍历

输入： 图 $G=(V,E)$，源节点 s

输出： 图 G 的广度优先遍历序列

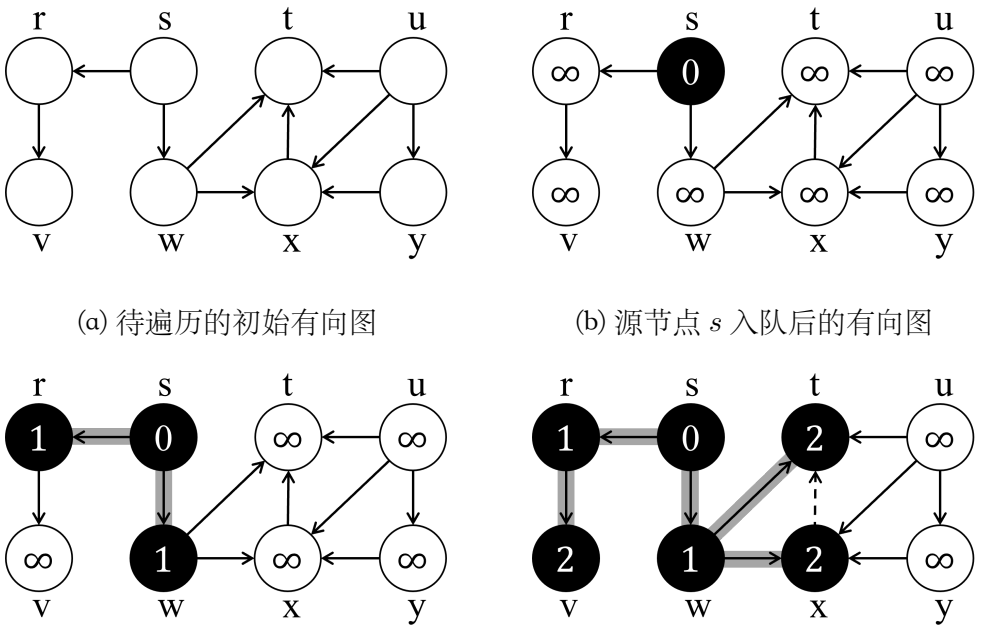
```
1:  function Bfs(G, s)
2:      for each vertex u ∈ V − {s} do
3:          u.color = WHITE
4:          u.d = ∞
5:      s.color =BLACK
6:      s.d = 0
7:      Q = ∅
8:      EnQueue(Q, s)
9:      while Q ≠ ∅ do
10:         u = DeQueue(Q)
11:         for each v ∈ Adj[u] do
12:             if v.color ==WHITE then
13:                 v.color =BLACK
14:                 v.d = u.d + 1
15:                 EnQueue(Q, v)
```

的等图操作，这些操作在本书的后续章节会经常用到。

1.3.1 图的遍历

图的遍历指对图的节点或边按特定的顺序进行访问，是对图最常见的操作操作之一，也是许多重要图算法的基础。按照对节点遍历的策略，遍历算法分为广度优先遍历和深度优先遍历算法。

广度优先遍历（Breadth First Search，BFS）是最常用的图遍历算法之一。给定图 $G=(V,E)$ 和一个源节点 s，广度优先遍历算法始终在广度方向上拓展已发现节点的边界，即算法在遍历完所有距离源节点 s 为 k 的节点后，再遍历距离源节点 s 为 $k+1$ 的节点，最终遍历完从源节点 s 可达的所有节



(a) 待遍历的初始有向图　(b) 源节点 s 入队后的有向图

(c) 节点 r 和 w 入队后的有向图　(d) 遍历完成后的有向图

图 1.2: 广度优先遍历算法在有向图上的运行过程

点。

算法1.5给出了广度优先遍历的一种实现。该算法为每个节点维护了两个额外属性：属性 $u.color$ 记录节点 u 的颜色，属性 $u.d$ 记录节点 u 到源节点 s 的距离（即经过的边的数量）。节点的颜色分为黑色 BLACK 和白色 WHITE，黑色节点表示该节点已被遍历，白色节点则表示该节点未被遍历。算法首先将除了源节点 s 之外的所有节点 u 涂上白色，并将的 $u.d$ 属性设置为初始值 ∞（第 2 到 4 行）。接着，算法将源节点 s 涂上黑色，并将 $s.d$ 属性设置为 0（第 5 到 6 行）。算法使用一个队列 Q 维护等待被遍历的节点，初始时，算法将源点 s 入队（第 7 到 8 行）。最后，算法从队列 Q 中不断取节点 u 进行遍历，并且把 u 还未被遍历的邻接点 v 入队（第 9 到 15 行），一直执行到队列 Q 空为止。

图1.2给出了广度优先遍历算法在示例有向图（图1.2a）上

的运行过程。图中添加了阴影的边表示算法执行过程中被遍历的边，节点 u 中的值表示该节点的距离属性值 $u.d$。初始时，算法设置节点 s 为黑色，到源节点的距离为 0（图1.2b）。随后，算法从节点 s 出发，遍历所有与该节点邻接、且未被访问过的节点，即节点 r 和 w，将这两个节点涂黑并设置到源节点的距离为 $0+1$ （图1.2c）。随后，算法分别继续从节点 r 和 w 出发对图进行广度优先遍历，从 r 节点向外遍历到节点 v，而从 w 节点向外遍历到节点 t 和 x（图1.2d）。最终，算法遍历完从源节点 s 出发可达的所有节点后，运行终止。

需要特别注意，进行广度优先遍历后，图 G 中可能仍然存在未被遍历的节点。例如，图 1.2d中的节点 u 和 y 都是从源节点 s 不可达的节点。在程序的控制流图中，不可达节点意味着节点不可能执行，因此实际上是*死代码*（Dead Code），可被死代码移除优化删去，从而简化程序的控制流图。

深度优先遍历（Depth First Search，DFS）是另外一种重要的图遍历算法。深度优先遍历总是对当前遍历的节点 u 的邻接点 v 进行遍历，直到节点 v 的所有邻接点都遍历完成为止。一旦节点 v 的所有邻接点均已遍历完成，深度优先遍历将会回溯到 v 的前驱节点 u 以继续遍历 u 剩余未被遍历的邻接点。该过程一直持续到从源节点可达的所有节点都被遍历为止。

算法1.6给出了对控制流图进行深度优先遍历的实现过程。算法为每个节点维护两个属性：属性 $u.color$ 记录节点 u 的颜色，而属性 $u.d$ 记录节点 u 被遍历的次序。节点的颜色 *color* 分为黑色 BLACK 和白色 WHITE，黑色表示该节点已被遍历，白色节点则表示该节点未被遍历。

算法 1.6 深度优先遍历

输入： 图 $G = (V, E)$，源节点 s

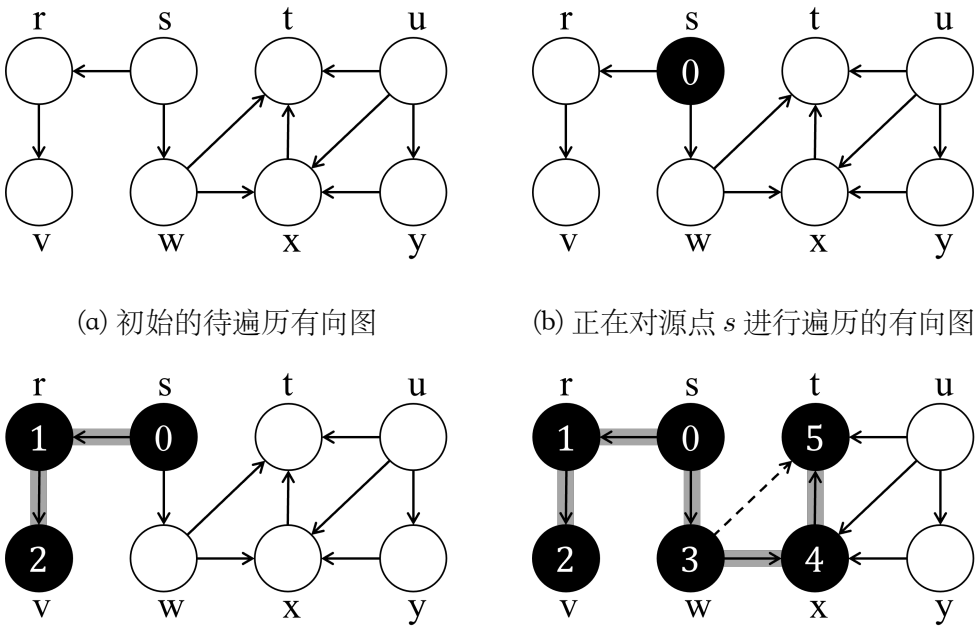
输出： 图 G 的深度优先遍历顺序

```
1:  time = 0
2:  function Dfs(G, s)
3:      for each vertex u ∈ V do
4:          u.color = WHITE
5:      DfsVisit(G, s)
6:  function DfsVisit(G, u)
7:      u.d = time
8:      time = time + 1
9:      u.color = BLACK
10:     for each vertex v ∈ Adj[u] do
11:         if v.color == WHITE then
12:             DfsVisit(G, v)
```

函数 Dfs 首先将所有节点涂成白色，并调用 DfsVisit 从初始节点 s 开始对图进行深度优先遍历。函数 DfsVisit 首先设置节点 u 的访问序号为全局计数器 $time$ 的值，并将节点 u 的颜色设置为黑色（即已访问）。接着，算法对节点 u 的每个邻接节点 v 进行检查，如果节点 v 为白色，算法递归遍历节点 v（第 11 到 13 行）。

作为示例，仍然考虑图1.3a所示的有向图。深度优先遍历算法从源节点 s 出发，调用函数 DfsVisit，对其进行遍历（图1.3b）；接着，算法依次遍历节点 s、r 和 v，在遍历过程中将其涂上黑色并设置其遍历序号（图1.3c）。在节点 v 遍历结束后，算法依次回溯到初始节点 s，并随后依次遍历节点 w、x 和 t。最后，算法遍历完从源节点 s 出发可达到的所有节点后，算法终止。

对于深度优先遍历算法，还有几个关键点需要注意。首先，从给定的源节点完成深度优先遍历后，图中可能仍然存



(a) 初始的待遍历有向图　(b) 正在对源点 s 进行遍历的有向图

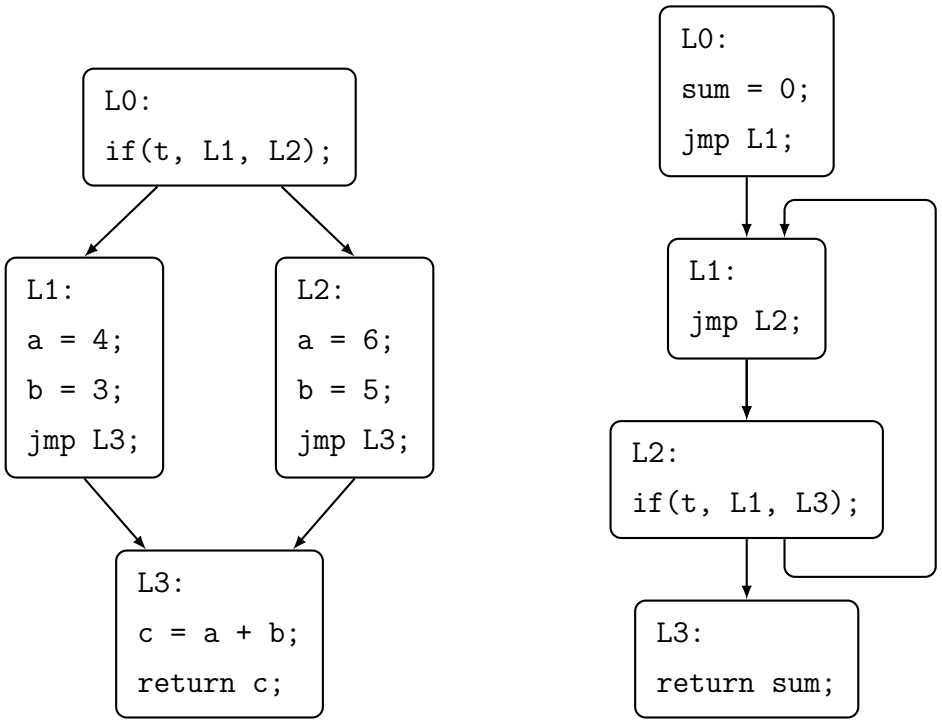
(c) 正在对节点 v 进行遍历的有向图　(d) 完成对所有节点遍历的有向图

图 1.3: 深度优先遍历算法在示例有向图上的运行过程

在未遍历到的节点。例如，图1.3d中的节点 u 和 y 就是从初始节点 s 出发未遍历的节点。和广度优先遍历算法类似，不可达节点是程序死代码，可被死代码移除优化删去。

其次，深度优先遍历的结果，依赖于函数 DfsVisit 中对当前节点 u 的邻接节点 v 的遍历顺序（算法第 10 行）。但不同的遍历结果，并不影响对控制流图的分析结果的正确性。

最后，深度优先遍历过程中，对有向图中有向边的不同处理时机，确定了对有向边的分类，可分为树边、前向边、后向边和交边。具体的，深度优先遍历过程中走过的有向边，称为**树边**（tree edge），所有的树边构成了图的**深度优先遍历生成树**（DFS spanning tree）。例如，图1.3中所有灰色阴影的边是树边，它们构成了以 s 为根节点的深度优先遍历生成树。从深度优先生成树的祖先节点指向孩子节点的边，称为**前向边**（Forward edge）；例如，图1.3中的边 (w, t) 即为一条前向边。从



(a) 一个包含分支跳转的控制流图

(b) 一个包含循环的控制流图

图 1.4: 控制流图示例

深度优先生成树的孩子节点指向祖先节点的边，称为*后向边*（Backward edge）。最后，有向图中其它所有边称为*交边*（Cross edge），它们由深度优先遍历生成树一棵子树中的节点，指向另外一棵子树的节点。

1.3.2 节点排序

对控制流图的分析算法（如数据流分析），往往需要按照特定的顺序对有向图的节点进行遍历。为了明确指定所需的遍历顺序，我们往往需要对控制流图中的节点进行线性排序。然而，由于程序控制流图中可能包括分支或者循环，使得节点之间一般并不存在严格的线性序。例如，图1.4a给定的控制流图包含分支跳转，尽管基本块 L_0 的直接后继为 L_1 和 L_2，

算法 1.7 伪拓扑排序

输入： 有向图 G、图的起始节点 s

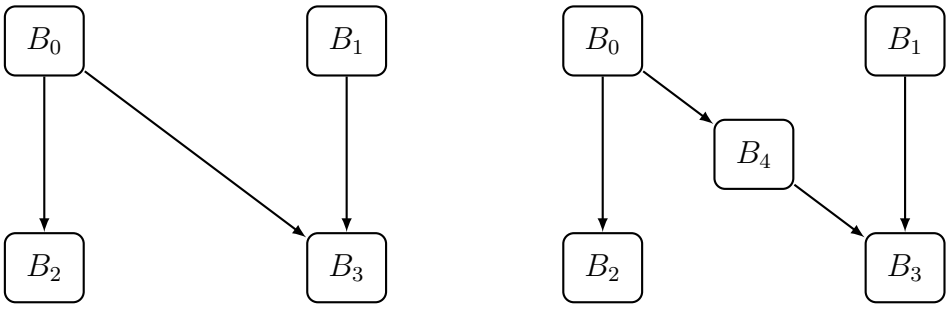
输出： 图的伪拓扑排序序列

```
1: blocks = []
2: function topoSort(G, s)
3:     Dfs(G, s)
4:     return rev(blocks)
5: function Dfs(G, u)
6:     for each successor v of u do
7:         if not visited v then
8:             Dfs(G, v)
9:     blocks = blocks ∪ {u}
```

但 L_1 和 L_2 之间并没有明确的线性顺序关系。在这种情况下，我们可以指定节点间的任意顺序，这样可以得到一种可能排序为 $L_0 \to L_1 \to L_2 \to L_3$。

为了实现对控制流图节点的线性排序，一种方法是利用有向图的*拓扑排序*（Topological Sort）。拓扑排序将有向图 $G = (V, E)$ 中的所有节点 V 排列为线性序列，该线性序列满足条件：如果图 G 包含有向边 (u, v)，则节点 u 在线性序列中处于节点 v 的前面。然而，拓扑排序只能处理*有向无环图*（Directed acyclic graph，DAG），而无法处理带有环状结构的控制流图。例如，图1.4b 给定的示例控制流图，包括一个环 $L_1 \to L_2 \to L_1$，则无法给定节点 L_1 和 L_2 的线性序。因此，在处理控制流图时，我们经常放宽对节点的线性排序的要求，仅保证尽可能多的节点都先于它们的后继出现。

基于上述思想，我们可以按照深度优先遍历的顺序，对控制流图进行*伪拓扑排序*（Quasi topological sort），其实现过程如算法 1.7所示。算法首先初始化一个链表 $blocks$ 用于记录已访问过的节点。接着，函数 topoSort 从初始节点 s 开始，对



(a) 包含关键边 (B_0, B_3) 的控制流图

(b) 通过加入新的节点 B_4，将关键边 (B_0, B_3) 切分为两条边 (B_0, B_4) 和 (B_4, B_3)

图 1.5: 关键边及其切分示例

有向图进行深度优先遍历，每当对一个节点 u 遍历结束时，会将该节点追加到 $blocks$ 的表尾。算法对有向图的深度优先遍历结束后，返回链表 $blocks$ 的逆序，作为伪拓扑排序结果（第 4 行）。

对图1.4中的示例控制流图应用算法 1.7，我们可得到图1.4a的一个可能的伪拓扑排序的序列为 $L_0 \to L_1 \to L_2 \to L_3$。同理，图1.4b的一个伪拓扑排序序列为 $L_0 \to L_1 \to L_2 \to L_3$。需要注意的是，由于对当前节点 u 的后继节点 v 访问顺序的不确定性（即算法1.7的第 6 行），图的伪拓扑排序的顺序一般也是不确定的。例如，图1.4a的另一种可能排序为 $L_0 \to L_2 \to L_1 \to L_3$。

1.3.3 关键边切分

编译器在对控制流图的分析和优化过程中，往往需要移动基本块中的语句，或者向基本块中增加新的语句。然而，控制流图结构特定的奇异性，会导致直接移动或添加语句将得到错误结果。

为具体理解可能出现的问题，我们考虑图1.5a所示的控制

算法 1.8 关键边切分

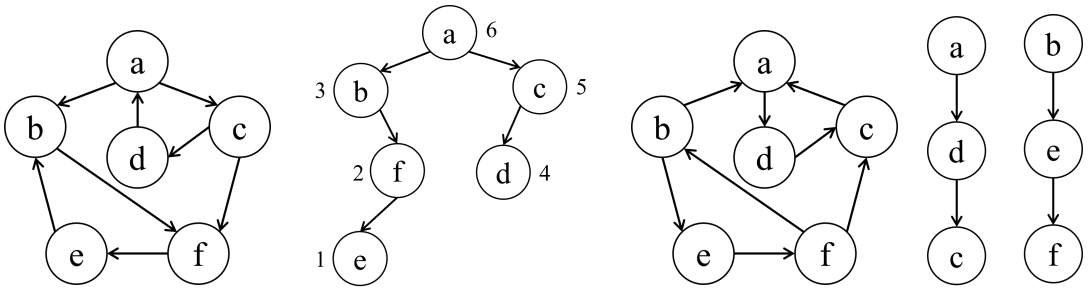
输入： 有向图 G

输出： 对关键边切分后的有向图

```
1: function criticalEdgeSplit(G)
2:     for each critical edge (u, v) ∈ G do
3:         // i.e., u has multiple successors, and v has multiple predecessors
4:         b = freshBlock()
5:         addVertex(G, b)
6:         removeEdge(G, u, v)
7:         addEdge(G, (u, b))
8:         addEdge(G, (b, v))
9:     return G
```

流图，假定我们需要新增某条语句 s，并且要求该语句在有向边 (B_0, B_3) 上执行。为此，一个自然的实现方式是将待新增的语句 s 放置在基本块 B_0 的末尾或 B_3 的开头，但不难发现，如果将语句 s 放在基本块 B_0 的末尾，则当执行流为 $B_0 \to B_2$，则新增的指令 s 将会错误地沿着边 (B_0, B_2) 多执行一次，如果语句 s 会产生副作用的话（如进行输出或内存写入等），则导致程序的可观察行为发生了变化。同理，不难验证，若把语句 s 放置在基本块 B_3 的开头，则该语句会沿着执行流 $B_1 \to B_3$ 多执行一次。

导致该问题的根因，是因为有向图1.5a中的边 (B_0, B_3) 是一条**关键边**（Critical edge），即该边的前驱节点 B_0 存在多个后继节点（包括节点 B_3），而同时该边的后继节点 B_3 存在多个前驱节点（包括节点 B_0）。为解决该问题，我们需要引入**关键边切分**（Critical edge splitting）操作，该操作通过在关键边上新增一个基本块，将关键边切分为两条非关键边。例如，为了对图1.5a的关键边 (B_0, B_3) 进行切分，我们新引入一个基本块 B_4，得到图1.5b中的结果。原来的边 (B_0, B_3) 被切分为



(a) 有向图中包括两个强连通分量 $\{a, c, d\}$ 和 $\{b, e, f\}$

(b) 有向图的深度优先遍历生成树，其中每个节点都标记了后序遍历编号

(c) 对原有向图的转置，按照后续遍历编号从大到小的顺序，依次进行深度优先遍历

(d) 遍历得到的每棵深度优先生成树，即是一个强联通分量

图 1.6: 计算有向图强连通分量的示例

两条新的边 (B_0, B_4) 和 (B_4, B_3)。这时，我们把待添加的语句 s，放置在新增的基本块 B_4 上，即可解决上述语句多执行一次的问题。

算法1.8给出了对给定的有向图 G，进行关键边切分的过程。对有向图的每条关键边 (u, v)，算法将新建一个空的基本块 b，并将 b 插入到图中。同时，算法移除原有的有向边 (u, v)，并加入两条新的有向边 (u, b) 和 (b, v)。

最后需要指出的是，关键边切分算法经常作为其它控制流图算法的预处理算法，即在进行其它算法前，先利用关键边切分算法对目标控制流图进行处理，移除关键边，进而保证待实施算法的正确性。

1.3.4 强连通分量

控制流图的分析和优化，经常需要分析控制流图包含的**强连通分量**（Strongly Connected Component，SCC）。例如，在进行**循环优化**（Loop Optimization）前，我们需要先确定控

算法 1.9 计算强连通分量

输入： 有向图 $G = (V, E)$

输出： 有向图 G 中的所有强连通分量

```
1: function Scc(G)
2:     SCC = {}
3:     call Dfs(G) to compute DFS spanning tree F
4:     number each vertex in F by post-order
5:     compute G^T
6:     S = sort all vertexs in descending order of post-order number
7:     for each u ∈ S do
8:         if not visited u then
9:             call Dfs(u) to compute the DFS spanning tree K
10:            SCC = SCC ∪ {K}
11:    return SCC
```

制流图中的循环结构，而循环结构的循环体往往由控制流图中的强连通分量组成。图1.6a给定的有向图中，由节点 a 进入的有向图节点子集 $\{a, c, d\}$ 构成了有向图的一个强连通分量。同理，节点集合 $\{b, e, f\}$ 构成了有向图的另一个强连通分量。

一般的，给定有向图 $G = (V, E)$，其强连通分量 $G_s = (V_s, E_s)$ 是一个包含最大节点的子图,对于任意一对节点 $u, v \in V_s$，都有从 u 到 v 的路径 p，且对路径上的任意一条边 $(f, t) \in p$，都满足 $(f, t) \in E_s$。直观上，强联通分量的任意两个节点间都互相可达，且可达路径上的所有边都属于该强连通分量。

算法1.9给出了计算强连通分量的一种方法。算法利用集合 SCC，来记录有向图 G 中包含的所有强连通分量。算法首先调用深度优先遍历算法 Dfs，得到图 G 的深度优先遍历生成树 F。（需要特别指出的是，对于一般有向图的深度优先遍历，将得到图 G 的深度优先森林，该森林由多棵独立的生成树构成。）接着，算法对深度优先生成树中的节点标号，按

照后序遍历顺序进行编号（第 4 行）。接下来，算法对有向图 G 进行转置，得到有向图 G^T，亦即 $G^T = (V, E^T)$，其中 $E^T = \{(v, u) \mid (u, v) \in E\}$。接着，算法对图 G 中节点，按照递减顺序进行排序，得到序列 S。接下来，算法从 S 中每个未访问过的节点 u 出发，进行深度优先遍历，得到的深度优先生成树 K，便是图 G 中的一个强连通分量，将 K 添加到集合 SCC 中。最后，算法返回 G 中所有的强连通分量 SCC，算法终止。

作为示例，考虑图1.6a中的有向图。首先，算法从节点 a 出发进行深度优先遍历，得到一棵以节点 a 为根节点的深度优先遍历生成树（图1.6b）。生成树的每个节点，都被标记了其对应的后序遍历编号。接着，算法将原有向图转置，得到新的有向图（即图1.6c）。接下来，算法按照后序遍历编号由大到小，对转置图节点进行遍历，并计算其深度优先遍历生成树。由于节点 a 的编号最大，因此先从 a 进行深度优先遍历后，得到节点集合 $\{a, d, c\}$，该集合的节点及相关的有向边，便是有向图 G 的一个强连通分量。同理，对剩余的节点进行深度优先遍历，可得到另一个强连通分量的节点集合 $\{b, e, f\}$。最终，算法计算得到如图1.6d所示的所有两个强连通分量。

1.4 编译优化引论

编译优化指编译器在编译过程中对程序代码进行的一系列变换和调整，以改进代码的质量。改进质量的可能指标包括提高代码的执行效率、减小代码体积或减少资源消耗，等等。基于本章前面讨论的控制流图，编译优化按照其操作的代码粒度，可划分为局部优化、全局优化和过程间优化。

在本小节，我们将结合几个典型优化算法，对这几种粒度的优化做全面讨论。特别的，本章给出的算法不依赖前置的程序分析过程，在后续章节，我们将讨论更复杂的依赖程序分析指导的优化。

1.4.1 局部优化

局部优化（Local optimization）的执行粒度是控制流图的单个基本块，而不考虑控制流图整体。在本小节，我们将讨论一种称为**局部值编号**（Local Value Numbering，LVN）的局部优化。

如果一个表达式在基本块中已经计算过，并且在该表达式再次出现时，其运算数没有被修改过，则称该表达式是**冗余的**。局部值编号优化试图在基本块中确定冗余表达式，并用之前计算过的值来替代冗余计算。例如，考虑如下基本块：

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$

第 4 条语句中的表达式 $a - d$ 是冗余的，因为该表达式在第 2 条被计算过，且其运算数 a 和 d 在第 2 条语句和第 4 条语句之间没有被修改过。相反，第 3 条语句中的表达式 $b + c$ 不是冗余的，因为第 2 条语句修改了其运算数 b，所以第 3 条语句出现的表达式 $b + c$ 的值，一般未必和第 1 条中出现的表达式 $b + c$ 的值相等。

对于冗余表达式，编译器可以通过进行**冗余删除**（Redundancy elimination）优化，重写该表达式，使其只计算一次。例

如，对上述基本块第 4 条语句中的冗余表达式 $a-d$，编译器可对代码做如下改写：

$$
\begin{aligned}
a &= b + c \\
t &= a - d \\
b &= t \\
c &= b + c \\
d &= t
\end{aligned}
$$

在重写的基本块中，第 2 条语句计算的表达式 $a-d$ 的值，先被赋值到一个全新变量 t 中，而后续的冗余计算 $a-d$，被重写为变量 t 值的直接读取。冗余删除可能会引入额外的赋值（如此处的语句 $b=t$），编译器后续阶段的优化可以分析确定赋值操作是否必要，并将不必要的赋值删除。

局部值编号算法依次遍历基本块中的每条语句，并为每个变量和表达式都分配一个唯一的编号。对于两个表达式，它们只有在操作符、以及所有操作数分别相等时，才会被分配相同的值编号。

算法1.10给出了局部值编号的过程。首先，算法引入哈希表 $H : e \mapsto v$，用于将表达式 e 映射到其唯一的值编号 v，该哈希表的初始值为空。我们用记号 $H[e]$ 代表哈希表的读操作。然后，算法依次遍历基本块中的每个语句 $x = y \oplus z$，并调用 lookupOrInsert 函数，从哈希表 H 中分别取得两个操作数 y 和 z 的值编号 v_y 和 v_z。如果 y 或 z 不在哈希表 H 中，函数 lookupOrInsert 会为其分配新的值编号，并将其插入到哈希表 H 中。接着，算法构造由值编号 v_y 和 v_z 构成的表达式 $\langle v_y, \oplus, v_z \rangle$，并以此表达式为哈希键值，在哈希表 H 中进行查找。如果找到该键相关的条目，则表示表达式已经计算过从

算法 1.10 局部值编号算法

输入： 基本块 B，包含 n 个形式为 $x_i = y_i \ \oplus_i \ z_i$ 的语句，$1 \leq i \leq n$

输出： 优化后的基本块

```
1: H = {} // H : e ↦ v
2: function lookupOrInsert(e)
3:     v = H[e]
4:     if v ≠ null then
5:         return (true, v)                    ▷ true indicates e ∈ H initially
6:     v = freshValueNumber()
7:     H[e] = v
8:     return (false, v)                       ▷ false indicates e ∉ H initially
9: function localValueNumber(B)
10:    for each statement x = y ⊕ z ∈ B do
11:        v_y = lookupOrInsert(y)
12:        v_z = lookupOrInsert(z)
13:        e = ⟨v_y, ⊕, v_z⟩
14:        (isRedunent, v) = lookupOrInsert(e)
15:        if isRedunent then
16:            Replace current statement with the variable associated with v
17:        H[x] = v
```

而是冗余的，则算法将当前表达式 $y \ \oplus \ z$ 替换为先前已被定义的变量。否则，如果没有找到条目，则表示这是此表达式在此块中的第一次计算，算法将为该表达式生成全新的值编号 v，并将该值编号插入哈希表 H 中。最终，算法将该值编号分配给赋值左侧的变量 x。

以上面讨论的基本块为例，使用局部值编号对其进行处理。算法首先处理语句 $a = b + c$，为变量 b 和 c 分配新的值编号 0 和 1，然后构造表达式 $b + c$ 的哈希键“$\langle 0, +, 1\rangle$”，并在哈希表中查找该哈希键。此时，算法未找到条目，则创建新条目“$\langle 0, +, 1\rangle$”并分配值编号 2，该值编号 2 也同时分配给赋值语句左侧的变量 a。同理，算法依次处理其他语句，最终可得

到如下结果：

$$
\begin{aligned}
a(2) &= b(0) + c(1) \\
b(4) &= a(2) - d(3) \\
c(5) &= b(4) + c(1) \\
d(4) &= a(2) - d(3)
\end{aligned}
$$

局部值编号算法确定了第 4 条语句中的冗余表达式 $a - d$（其值编号为 $\langle 2, -, 3\rangle$），并可对其进行优化。

对局部值编号算法，有几个关键点需要注意。首先，上述给定的局部值编号算法主要处理了二元操作 $\oplus$（如加、减、乘、除等）。但实际上，该算法也可以扩展到处理一元操作或其它操作，只需调整值编号表达式 $\langle v_y, \oplus, v_z\rangle$ 的构造方法。

其次，局部值编号算法在处理涉及函数调用的优化时较为受限。因为函数调用可能会改变全局状态或具有副作用，使得值编号失效。为了处理这种情况，需要进行过程间分析，来明确函数调用的语义。

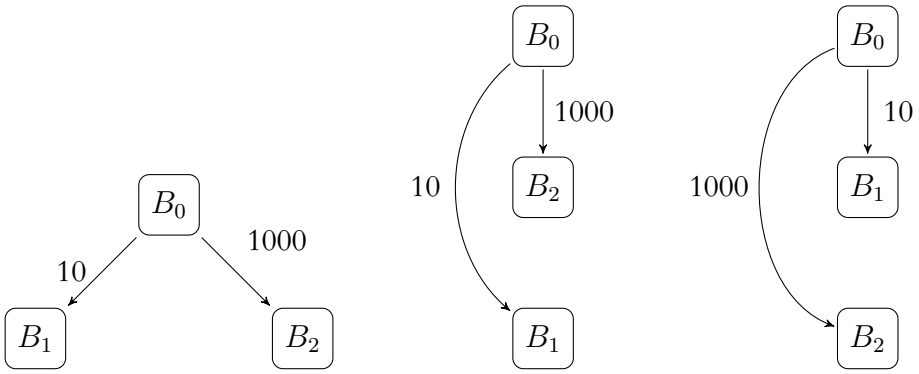
最后，局部值编号优化可以和其它优化技术结合使用，以进一步提高局部值编号的优化效果。可以和值编号技术结合使用的优化技术包括交换运算、常量折叠、代数恒等式化简，等等。首先，如果表达式中的运算符满足交换性质，则我们可以通过交换表达式中运算数的顺序，使得表达式具有唯一的值编号。例如，表达式 $a + b$ 和 $b + a$ 的语法形式不同，因此其分别具有值编号 $\langle a_v, +, b_v\rangle$ 和 $\langle b_v, +, a_v\rangle$，值编号的不同导致局部值编号算法无法将其判定为冗余计算。但由于加法 $+$ 满足交换律，则我们可以通过对表达式中操作数进行重新排序，来给这两个表达式分配相同的值编号。一般的，编译器在执行局部值编号前，可先对满足交换律的运算数进行排

序，以期检测更多的等价表达式，从而消除更多的冗余计算。其次，局部值编号可利用常量折叠，将常量表达式直接计算出来，不但能减少运行时的计算量，还可能发现潜在的冗余表达式。例如，在为表达式分配值编号时，如果表达式的操作数都是常量（如表达式 $3+4$），则编译器可直接计算其结果（即 7），并将结果作为常量存储。最后，局部值编号还可以应用代数恒等式的性质（如 $x+0=x$ 和 $x*1=x$ 等），来对表达式进行化简，为此，编译器需要在执行局部值编号前，对表达式做额外的语法形式检查，并按语法制导的规则对等式进行化简。

1.4.2 全局优化

全局优化（Global optimization）也称**过程内优化**（Intra-procedural optimization），它对整个函数控制流图进行分析和优化。由于其比局部优化考虑了更大的程序粒度，所以往往能达到更好的优化效果。在本小节，我们将重点讨论一种称为**全局基本块放置**（Global block placement）的全局优化。全局基本块放置是通过按特定的顺序放置函数的基本块，从而减少分支预测失败的可能性，优化指令缓存的使用，使得代码的执行更为高效，提高整体执行性能。

许多现代处理器在执行分支指令时，其分支执行成本是非对称的。由于指令缓存的存在，默认执行路径（fall-through）中的代码往往已经被预取到缓存中，因此其执行成本通常低于跳转路径（taken branch）。因此，编译器可以通过将高频执行的基本块放置在默认执行路径上，并将低频执行的基本块放置在挑战路径上，从而可以充分利用这种非对称的分支执行成本。



(a) 基本块 B_0 的两个分支的执行频率分别为 10 和 1000

(b) 将基本块 B_2 放置在 B_0 后（即默认执行路径上），而将基本块 B_1 放置在跳转路径上

(c) 将基本块 B_1 放置在 B_0 后（即默认执行路径上），而将基本块 B_2 放置在跳转路径上

图 1.7: 带有执行频率信息的控制流图示例

如图1.7a所示，基本块 B_0 有两个后继基本块 B_1 和 B_2，其中路径 $B_0 \to B_2$ 和 $B_0 \to B_1$ 的执行次数分别为 1000 和 10（亦即 $B_0 \to B_2$ 的执行频率比 $B_0 \to B_1$ 高 100 倍）。则在这种情况下，编译器应该将频率较高的基本块 B_2 放置在紧邻 B_0 后（如图1.7b所示），而将基本块 B_1 放置在远离 B_0 的跳转路径上，从而比另外可能的放置方式相比（如图1.7c所示），减少总的分支执行成本。

全局基本块放置优化包含分析和转换两个主要阶段。分析阶段主要确定每个基本块的执行信息，从而为后续的基本块放置提供依据。为了收集执行信息，编译器可以使用剖析工具（profiler），来收集程序执行期间的运行时信息。常见的收集剖析信息的方法有：（1）基于插桩（instrumentation）的方法。该方法通过在程序中插入额外的代码来收集程序执行数据，这些执行数据包括函数调用和执行时间、基本块的执行次数，等等。这种方法提供的数据比较详细，但由于需要修改或增加程序代码，因此会增加程序运行时开销，影响程序

算法 1.11 热路径构建算法

输入： 控制流图 G

输出： 热路径链集合

```
1:  function buildHotPaths(G)
2:      Chains = {}
3:      for each basic block b ∈ G do
4:          d = {b}
5:          priority(d) = +∞
6:          Chains = Chains ∪ {d}
7:      P = 0
8:      for each edge (x, y), in descending order of frequency do
9:          if x is the tail of chain a and y is the head of chain b then
10:             c = {a} ∪ {b}
11:             Chains = Chains − {a} − {b} ∪ {c}
12:             priority(c) = min(priority(a), priority(b))
13:             if priority(c) = +∞ then
14:                 priority(c) = P
15:             P = P + 1
16:     return Chains
```

性能。（2）基于采样（sampling）的方法。该方法通过定期中断程序的执行并记录状态来收集程序执行数据，通常，该方法可以利用操作系统的时钟中断来实现。虽然该方法对程序性能影响较小，但采样数据精确度依赖于采样频率，且信息粒度较粗。（3）基于事件（event）的方法。该方法通过捕捉特定事件来收集执行信息数据，特定事件可能包括内存分配或系统调用、分支跳转等。通常该方法对程序性能影响较小，但许多事件的获取往往需要底层硬件的特殊支持（如性能计数器等），因此监控事件的种类有限且受可用硬件的限制。

转换阶段依据分析阶段收集得到的执行信息，来构建控制流图中的热路径，最终确定基本块的布局。热路径是指最频繁执行边构成的路径，我们使用算法1.11给出的贪心策略来

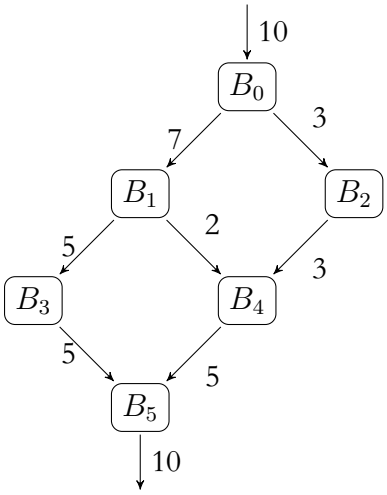


图 1.8: 标记了执行频率的示例控制流图

确定热路径。算法首先创建每个基本块 B 的退化链（即只包含该块 B 本身），并为每个退化链设置初始优先级 $+\infty$（第 2 到 6 行）。然后，算法按图中有向边执行频率降序的顺序，依次处理控制流图中的每条边 (x, y)，如果 x 是某条链 a 的尾节点且 y 是某条链 b 的首节点，则合并这两个链 a 和 b 为一条新的链 c。当合并两个链 a 和 b 时，新链 c 的优先级 $priority$ 需重新设置为被合并链 a 和 b 中优先级的最小值，这确保了新链 c 的优先级不会超过它所包含的任何一个原始链 a 和 b 的优先级。如果合并的是两个退化链（即优先级为 $+\infty$），则新链的优先级设置为当前合并操作的次数 P。

作为示例，我们考虑图1.8中给定的标记了执行频率的控制流图，应用上面的算法构建热路径，结果在表1.1中给出。最终，我们得到了两条热路径的链 $(B_0, B_1, B_3, B_5)_0$ 和 $(B_2, B_4)_3$，其下标 0 和 3 分别表示两条链的优先级。

在确定热路径链后，我们接下来需要将基本块按照这些路径进行排列，该过程由算法1.12给出。该算法使用了工作表思想，即算法依次将每个热路径链 t 加入工作表 W 进行处理。在初始时，算法首先将包含控制流图 G 入口节点的链 t 放入

表 1.1: 热路径的构建过程

边	链	P
–	$(B_0)_\infty, (B_1)_\infty, (B_2)_\infty, (B_3)_\infty, (B_4)_\infty, (B_5)_\infty$	0
(B_0, B_1)	$(B_0, B_1)_0, (B_2)_\infty, (B_3)_\infty, (B_4)_\infty, (B_5)_\infty$	1
(B_3, B_5)	$(B_0, B_1)_0, (B_2)_\infty, (B_3, B_5)_1, (B_4)_\infty$	2
(B_4, B_5)	$(B_0, B_1)_0, (B_2)_\infty, (B_3, B_5)_1, (B_4)_\infty$	2
(B_1, B_3)	$(B_0, B_1, B_3, B_5)_0, (B_2)_\infty, (B_4)_\infty$	3
(B_0, B_2)	$(B_0, B_1, B_3, B_5)_0, (B_2)_\infty, (B_4)_\infty$	3
(B_2, B_4)	$(B_0, B_1, B_3, B_5)_0, (B_2, B_4)_3$	4
(B_1, B_4)	$(B_0, B_1, B_3, B_5)_0, (B_2, B_4)_3$	4

工作表 W 中。在每轮循环中，算法从工作表 W 中移除优先级最高（即优先级 $priority$ 数值最低）的链 c，并将该链中的每个基本块 x 按顺序放置在可执行代码的末尾。然后，算法检查链 c 中每个基本块 x 在控制流图 G 中的后继节点 y，如果该节点 y 还未被放置过，则找到该边 (x, y) 所属的链 t，如果此时 t 不在工作表 W 中的话，则将链 t 加入到工作列表 W 中。算法一直循环进行，直到工作表 W 空为止。

作为示例，我们将该算法应用于表1.1中的热路径链。首先，算法将链 $(B_0, B_1, B_3, B_5)_0$ 加入工作表 W，则算法依次将基本块 B_0、B_1、B_3 和 B_5 加入到可执行代码末尾。接着，算法沿着有向边 $\langle B_0, B_2\rangle$ 考虑基本块 B_0 的后继块 B_2，由于该基本块 B_2 还未被放置到最终代码中，因此算法将有向边 $\langle B_0, B_2\rangle$ 所在的路径链 $(B_2, B_4)_3$ 加入工作表，并继续进行下一轮循环。最终，当算法执行结束时，生成的代码布局为 $(B_0, B_1, B_3, B_5, B_2, B_4)$。

需要注意的是，由于控制流图中可能包含执行频率相同的边，因此可能会得到不同的热路径链，进而产生不同的代码布局。例如，图1.8中的两条边 (B_3, B_5) 和 (B_4, B_5) 有相同的权

算法 1.12 代码布局算法

输入：热路径链集合

输出：排列后的基本块

```
1: function blockLayout(G, Chains)
2:     t = chain containing the entry node of G
3:     W = {t}
4:     while W ≠ ∅ do
5:         c = Remove a chain with the highest priority from W
6:         for each basic block x in chain c do
7:             Place x at the end of the executable code
8:         for each basic block x in chain c do
9:             for each successor block y of x do
10:                if y is not yet placed then
11:                    t = chain containing edge (x, y)
12:                    if t ∉ W then
13:                        W = W ∪ {t}
```

重 5，假如我们在生成热路径时，先处理 (B_4, B_5)，再处理 (B_3, B_5)，那么它将生成两个路径链 $(B_0, B_1, B_3)_0$ 和 $(B_2, B_4, B_5)_1$，从而生成最终的代码布局 $(B_0, B_1, B_3, B_2, B_4, B_5)$. 因此，在实际应用中，我们可以根据具体需求调整优先级的策略，以达到最佳的代码优化效果。

1.4.3 过程间优化

过程间优化（Inter-procedural optimization）在优化过程中同时考虑多个过程（也称函数或方法），因此具有比局部和全局优化更大的优化粒度。在本小节，我们将重点讨论一种重要的过程间优化— 内联替换（Inline substitution）优化。

内联替换优化的目标是通过将被调用函数体替换到该函数调用点，从而消除函数调用的开销，提升程序性能。函数是编程语言的重要抽象，保证了模块抽象和代码复用，但是

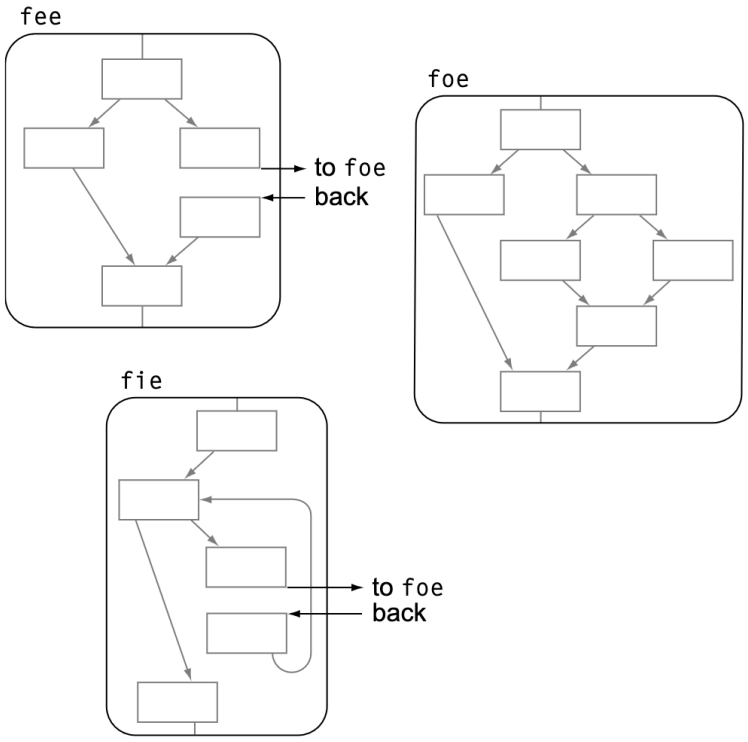


图 1.9: 内联替换之前的函数调用

函数调用和返回通常涉及一系列复杂的底层操作，包括分配活动记录、计算函数实参、保存调用者执行状态、创建被调用者环境、控制转移、返回值传递、以及控制返回，等等。这些操作导致了函数调用和返回，产生可观的运行时开销。而通过内联替换优化，编译器不但可以消除这些由于函数调用返回导致的运行时开销，还能为被调用的函数体代码提供更丰富的上下文信息（注意到这些信息本来是处于调用函数内，因此对内联前的被调用函数不可见），从而提供更多的潜在优化机会。

内联替换的优化过程涉及对函数调用点的重写，以及对参数传递过程的适当修改。具体来说，编译器首先将被调用函数的函数体代码复制到调用点，然后处理函数调用中实参和形参的绑定，将调用点处的实际参数值映射到被调用函数中的形参。例如，对于函数调用

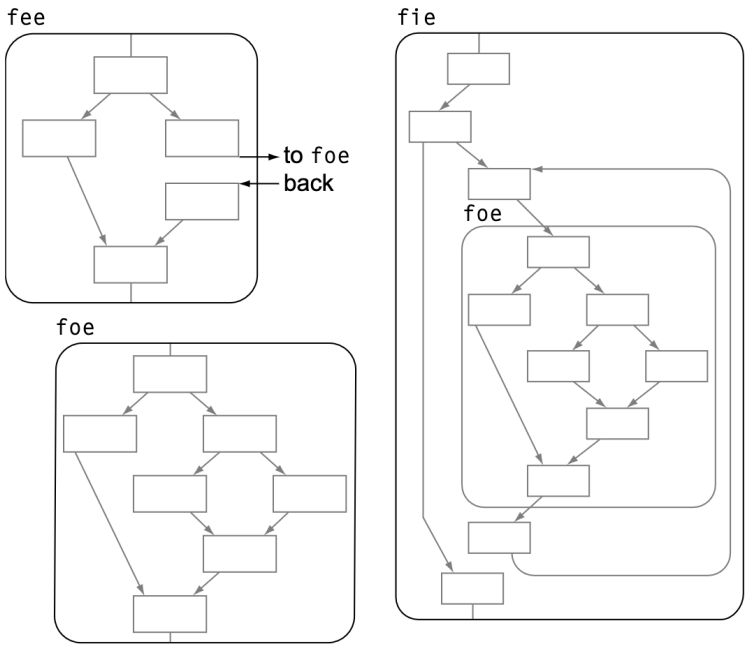


图 1.10: 内联替换之后的控制流图

```
y = f(e_1, ..., e_n);
```

假设其中被调用函数 f 的函数体为

```
int f(int x_1, ..., int x_n) {S; return b; }
```

则经过被函数代码复制以及参数绑定后，产生的代码为

```
x_1 = e_1; ...; x_n = e_n;
S;
y = b;
```

在进行内联替换优化时，有几个关键点需要考虑。首先，在控制流图数据结构上实现内联替换时，编译器需要修改调用函数的控制流图结构。具体的，编译器需要连接调用点的前驱基本块到被调用函数的入口块，并连接被调用函数体的出口块到调用点的后继基本块，以确保程序控制流图的结构合

法性。例如，考虑图1.9中的控制流图，函数 `fee` 和 `fie` 都调用了函数 `foe`。在对被调用函数 `foe` 完成内联替换后，得到的结果控制流图如图1.10所示，其中函数 `fie` 中对函数 `foe` 的调用已经被内联，而函数 `fee` 的调用则没有（我们下面将讨论内联策略，它决定是否对特定的调用点函数进行内联）。

其次，编译器需要重命名被内联函数中的局部变量（包括形参）。由于被内联的函数中的局部变量和形参，可能会和内联到的函数中的局部变量重名，造成名字冲突。为此，编译器需要在内联前，对被内联函数局部变量和形参进行重命名。另外，考虑到被内联函数可能被内联替换到同一个函数内的多个不同调用点，因此变量重命名操作需要在每个调用点处重复进行。

最后，编译器需要根据优化目标，确定内联策略，以决定在哪些函数调用点进行内联替换优化。尽管内联替换优化可以通过消除函数调用的开销提高程序性能，但它也会带来负面作用。尤其是，由于被内联的函数代码在调用点被复制多份，可能导致代码体积增大、局部变量增多（进而导致更多的变量干涉）等问题。因此，编译器需要综合考虑调用者、被调用者以及调用点的各种因素，确定在每个调用点是否进行内联的策略。影响编译器决策的因素包括但不限于：（1）调用者或被调用者的体积大小：如果被调用者体积小于给定阈值，则内联可能减少最终的代码体积；（2）静态或动态调用次数：只对静态频繁出现或动态频繁进行的调用点进行优化；例如，在图1.10的示例中，由于函数 `fee` 只调用了函数 `foe` 一次，因此编译器决定不进行内联，而保留原来对函数 `foe` 的调用；（3）常量值的实参：只对具有常量值实参数的函数进行内联，从而创造内联后进一步进行常量传播优化的机会；（4）参数数

量：对多参数函数内联可能会减少参数传递的代价；（5）叶函数：优先内联叶函数，即不含其它调用的函数，这些函数时调用图中的叶子节点；（6）调用的位置：对循环中的调用点进行内联；如图1.10 中，函数 `fie` 对函数 `foe` 的调用被进行了内联；（7）动态执行剖析：基于动态剖析得到的执行数据，优先内联执行时间占比高的函数。

在实际实现中，编译器一般可以预先计算上述指标中的部分或全部，然后应用一个或一组启发式方法，来确定哪些函数调用点需要内联。编译器所采用的启发式方法，需要根据被编译的语言或优化目标来确定。

1.5 本章小结

控制流图是编译器常用的重要中间表示，为程序分析和优化提供了基础。本章首先讨论了控制流图的定义，以及从高层表示生成控制流图的自顶向下方法，和从底层表示生成控制流图的自底向上方法。接着，本章讨论了对控制流图的基本操作，包括遍历算法、节点排序、关键边切分和强连通分量等。最后，控制流图非常自然的刻画了基于控制流图的程序分析和优化的粒度，本章具体讨论局部优化（如局部值编号）、全局优化（如全局基本块放置）、和过程间优化（如内联替换）等，为后续章节讨论更复杂的程序分析和优化奠定基础。

1.6 深入阅读

控制流图是一种经典的编译器中间表示，它最早于 1970 年由 Frances E. Allen[3] 所提出。许多编译器论文和教材 [2, 6,

14, 20, 10, 3] 对其都全面讨论了控制流图。目前许多生产级的编译器实现（如 GCC [1] 或 LLVM [19]），都使用了控制流图或其变种，作为程序分析和优化的核心中间表示。

Robert Tarjan[25] 于 1972 年首次提出深度优先搜索，并讨论了其在图操作中的重要作用，随后又提出强连通分量的算法。许多算法相关书籍 [17, 18, 24, 11, 23] 都详细讨论了图相关的算法。Briggs 等 [7] 于 1998 年讨论了关键边，并给出了关键边切分的算法。

基于控制流图的编译优化技术在编译器设计中扮演着重要角色，许多编译器的参考书及文献 [20, 10] 都对这些技术进行了深入探讨。局部值编号技术最早由 Balke 在 20 世纪 60 年代提出，并逐步扩展到包括代数简化和常量折叠等优化技术 [4, 8]。此外，Ershov[13] 在更早期的系统中也实现了类似的效果。

全局代码块放置算法的研究始于 Pettis 和 Hansen 的工作 [21]，他们提出了在全局和整个程序范围内优化代码块放置的方法。这一领域的后续研究主要集中在如何收集更准确的性能数据以及改进代码放置策略 [15, 16]。内联替换作为一种常见的优化技术，已在编译器领域被广泛讨论了数十年 [4]。尽管内联替换的实现相对简单，但对其优化效果和获利的研究一直是重点 [5, 9, 12, 22, 26]。

1.7 思考题

1. 根据控制流图的定义，使用一种你熟悉的编程语言，给出控制流图的数据结构实现。

2. 给定如下一段代码，使用自顶向下或自底向上的方式为它构建控制流图。

```
int sum(int n){
    int s = 0, i = 0;
    while(i <= n){
        if (i % 2){ s += i; }
        else { s -= i; }
        i += 1;
    }
    return s;
}
```

3. 算法1.2中并没有给出对 `for` 循环语句的处理，尝试扩展该算法使其能够处理 `for` 循环。如果 `while` 循环或 `for` 循环中存在 `break` 或 `continue` 语句，该如何进行处理?

4. 对于图1.11中的控制流图，给出图中基本块的一个伪拓扑排序。并计算该图的强连通分量。

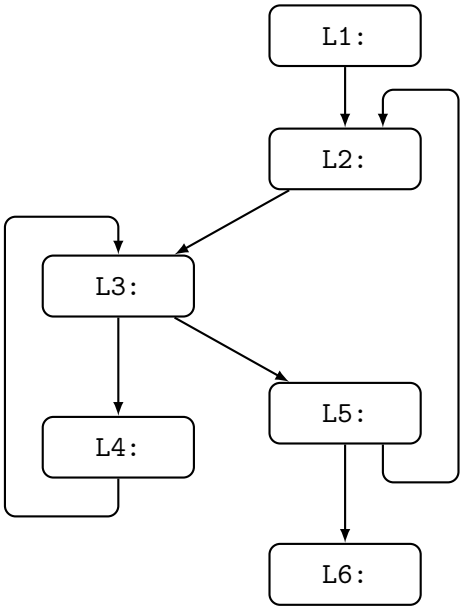


图 1.11: 控制流图

5. 关键边切分需要找到图中的关键边。在1.3.3节中，我们给出了关键边的定义，请给出寻找控制流图中关键边的算法。利

用你的算法，判断图1.11中是否存在关键边?

6. 为了计算图的强连通分量，算法1.9在第一次深度优先遍历后，对深度优先森林中的节点按后序遍历顺序标号。思考能否去除该步骤？为什么?

7. 局部值编号算法（算法1.10）依赖于哈希表来记录表达式的值编号。在处理过程中，如果哈希表的冲突处理不够高效，可能会影响算法的性能。思考如何设计符号值的数据结构，并结合优化哈希表的冲突处理机制，以提高算法的效率。

8. 对图1.12中的控制流图，给出应用全局基本块放置算法的结果。

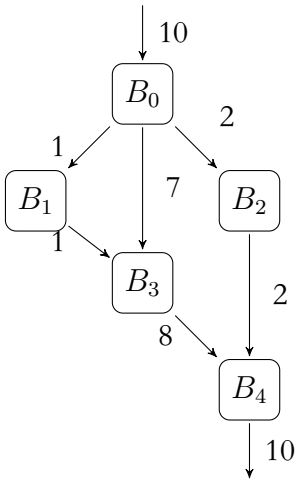


图 1.12: 控制流图

9. 结合1.4.3小节，简述内联替换对程序执行效率的潜在影响，并讨论如何在编译过程中平衡这种优化的优缺点。

参考文献

[1] Control flow (gnu compiler collection (gcc) internals).

[2] Alfred V. Aho and Alfred V. Aho, editors. Compilers: Principles, Techniques, & Tools. Pearson/Addison Wesley, 2nd ed edition.

[3] Frances E Allen. Control flow analysis. 5(7):1 – 19.

[4] Frances E Allen and John Cocke. A catalogue of optimizing transformations.

[5] J Eugene Ball. Predicting the effects of optimization on a procedure body. 14(8):214 – 220.

[6] William A Barrett, Rodney M Bates, David A Gustafson, and John D Couch. Compiler Construction: Theory and Practice. SRA School Group.

[7] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. 28(8):859 – 881.

[8] John Cocke. Global common subexpression elimination. 5(7):20 – 24.

[9] Keith D Cooper, Mary W Hall, and Linda Torczon. An experiment with inline substitution. 21(6):581 – 601.

[10] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Elsevier/Morgan Kaufmann, 2nd ed edition.

[11] Thomas H. Cormen, editor. Introduction to Algorithms. MIT Press, 3rd ed edition.

[12] Jack W Davidson and Anne M Holler. A study of a c function inliner. 18(8):775 – 790.

[13] Andrei P. Ershov. On programming of arithmetic operations. 1(8):3 – 6.

[14] Charles N Fischer and Richard J LeBlanc Jr. Crafting a Compiler with C. Benjamin-Cummings Publishing Co., Inc.

[15] Nikolas Gloy and Michael D Smith. Procedure placement using temporal-ordering information. 21(5):977 – 1027.

[16] Amir H Hashemi, David R Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. 32(5):171 – 182.

[17] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. Computer Algorithms. Computer Science Press.

[18] Jon Kleinberg and Éva Tardos. Algorithm Design. Pearson/Addison-Wesley.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04).

[20] Steven Muchnick. Advanced Compiler Design Implementation. Morgan kaufmann.

[21] Karl Pettis and Robert C Hansen. Profile guided code positioning. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, pages 16 – 27.

[22] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. 32(3):137 – 142.

[23] Robert Sedgewick and Kevin Daniel Wayne. Algorithms. Addison-Wesley, 4th ed edition.

[24] Steven S. Skiena. The Algorithm Design Manual. Springer, 2nd ed edition.

[25] Robert Tarjan. Depth-first search and linear graph algorithms. 1(2):146 – 160.

[26] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 977 – 989. ACM.