

6. 中间代码优化

本章实验为**实验五**，任务是在词法分析、语法分析、语义分析和中间代码生成程序的基础上，使用数据流分析算法等消除效率低下和无法被轻易转化为机器代码的中间代码，从而将C—源代码翻译成的中间代码转化为语义等价但是更加简洁高效的版本。本实验是**实验三**（本书第四章，下同）的延续，将针对生成代码的语义一致性及代码的简洁性和高效性进行检查。

需要注意的是，由于本次实验的代码会与之前实验中你已经写好的代码进行对接，因此保持一个良好的代码风格、系统地设计代码结构和各模块之间的接口对于整个实验来讲相当重要。

6.1 实验内容

6.1.1 实验要求

在本次实验中，对于给定的程序，在实验三生成的中间代码IR基础上，编译器对IR进行中间代码优化。实验假设同实验三，输入的源程序不包含词法语法语义分析错误，可以正常生成IR。对于给定的IR，你的程序要能够进行如下的中间代码优化：公共子表达式消除、无用代码消除、常量传播、循环不变表达式外提、强度削减、控制流图优化等。

- 1)**公共子表达式消除**：避免重复计算已经在先前计算过的表达式。
- 2)**无用代码消除**：删除执行不到的基本块和指令、仅存储但不使用的变量等。
- 3)**常量传播**：对于值恒为常量的表达式进行常量替换。
- 4)**循环不变表达式外提**：每次执行结果都不变的表达式，可移动到循环外部。
- 5)**强度削减**：将高代价运算转换为低代价的运算。
- 6)**控制流图简化**：去除空基本块，优化执行顺序。

实验五主要考察程序输出的中间代码的执行效率，因此需要考虑如何优化中间代码的生成，基于后文所述的中间代码优化方法进行代码优化。虚拟机将根据优化之后的代码行数与操作数量评估优化效果。

6.1.2 输入格式

你的程序的输入是一个实验三输出的中间表达文本文件。输入文件每行一条中间代码，如果包含多个函数定义，则通过FUNCTION语句将这些函数隔开。程序需要能够接收一个输入文件名和一个输出文件名作为参数。例如，假设程序名为cc、输入文件名为input.ir、输出文件名为output.ir、程序和输入文件都位于当前目录下，那么在Linux命令行下运行./cc input.ir output.ir即可获得以input.ir作为输入文件的输出结果。

6.1.3 输出格式

实验五在实验三的基础上进行中间代码优化，输出格式与实验三相同。我们将使用虚拟机小程序统计优化后的中间代码所执行过的各种操作的次数，以此来估计程序生成的中间代码的效率。

6.1.4 测试环境

程序将在如下环境中被编译并运行（同实验一）：

- 1)GNU Linux Release: Ubuntu 20.04, kernel version 5.4.0-28-generic;
- 2)GCC version 7.5.0;
- 3)GNU Flex version 2.6.4;
- 4)GNU Bison version 3.0.4.
- 5)虚拟机(基于Python3.8实现)。

一般而言，只要避免使用过于冷门的特性，使用其它版本的Linux或者GCC等，也基本上不会出现兼容性方面的问题。注意，实验五的检查过程中不会去安装或尝试引用各类方便编程的函数库（如glib等），因此请不要在你的程序中使用它们。在实验报告中，请标注所使用的版本。

6.1.5 提交要求

实验五要求提交如下内容（同实验一）：

- 1)Flex、Bison以及C语言的可被正确编译运行的源程序。
- 2)一份PDF格式的实验报告，内容包括：
 - a)程序实现了哪些功能？简要说明如何实现这些功能。清晰的说明有助于助教对程序所实现的功能进行合理的测试。

b)程序应该如何被编译？可以使用脚本、makefile或逐条输入命令进行编译，请详细说明应该如何编译程序。无法顺利编译将导致助教无法对程序所实现的功能进行任何测试，从而丢失相应的分数。

c)实验报告的长度不得超过三页！所以实验报告中需要重点描述的是程序中的亮点、最个性化、最具独创性的内容，而相对简单的、任何人都可以做的内容则可不提或简单地提一下，尤其要避免大段地向报告里贴代码。实验报告中所出现的最小字号不得小于五号字（或英文11号字）。

6.1.6 样例（必做内容）

实验五的样例包括**必做内容样例与选做要求样例**两部分，分别对应于实验要求中的必做内容和选做要求。请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。这节列举必做内容样例。

样例1（局部公共子表达式消除）：

输入：

```
1 FUNCTION main :  
2 READ t1  
3 v1 := t1  
4 READ t2  
5 v2 := t2  
6 t3 := v1 + #2  
7 v3 := t3  
8 t4 := v3 + #1  
9 v4 := t4  
10 t5 := v1 + #3  
11 v5 := t5  
12 t6 := v1 + v2  
13 v6 := t6  
14 t7 := v1 + #2  
15 v7 := t7  
16 t8 := v1 + #2  
17 v2 := t8  
18 t9 := #0  
19 RETURN t9
```

输出：

对于这段只包含单个基本块的中间代码，对基本块中表达式构造有向无环图，并且运用代数恒等式转换。我们检查有向无环图中一个节点N，是否与另一个节点M具有相同的运算符与子节点，则消除公共子表达式，生成的中间代码可以是这样的：

```
1 FUNCTION main :  
2 READ t1
```

```
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := v1 + #2
7 v3 := t3
8 t4 := v3 + #1
9 v4 := t4
10 t5 := v1 + #3
11 v5 := t5
12 t6 := v1 + v2
13 v6 := t6
14 v7 := t3
15 v2 := t3
16 t9 := #0
17 RETURN t9
```

如果你的方法足够聪明，你会发现t4与t5也是相同的，通过设计合适的框架，能够使得公共子表达式的消除效果更好。

样例2（局部无用代码消除）：

输入：

```
1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 v3 := #2
7 v4 := #3
8 t3 := v2 + v3
9 v1 := t3
10 t4 := v1 - v4
11 v2 := t4
12 t5 := v2 + v3
13 v3 := t5
14 t6 := v1 - v4
15 v4 := t6
16 t7 := v1 + v3
17 v5 := t7
18 t8 := v1 - v4
19 v6 := t8
20 t9 := v4 + v6
21 v7 := t9
22 t10 := v2 + v1
23 v1 := t10
24 t11 := v7 + v6
25 v8 := t11
26 WRITE v8
27 t12 := v1 + v4
28 v9 := t12
29 WRITE v9
30 t13 := #0
31 RETURN t13
```

输出：

对于v1-v7七个变量，构造有向无环图。我们基于有向无环图，迭代地去除没有父节点且未被使用的根节点，消除无用代码，因此生成的中间代码可以是这样的：

```
1 FUNCTION main :  
2 READ t1  
3 v2 := t1  
4 v3 := #2  
5 v4 := #3  
6 t2 := v2 + v3  
7 v1 := t2  
8 t3 := v1 - v4  
9 v2 := t3  
10 t4 := v2 + v3  
11 v3 := t4  
12 t5 := v1 - v4  
13 v4 := t5  
14 t6 := v1 - v4  
15 v6 := t6  
16 t7 := v4 + v6  
17 v7 := t7  
18 t8 := v2 + v1  
19 v1 := t8  
20 t9 := v7 + v6  
21 v8 := t9  
22 WRITE v8  
23 t10 := v1 + v4  
24 v9 := t10  
25 WRITE v9  
26 t11 := #0  
27 RETURN t11
```

样例3（常量折叠）：

输入：

```
1 FUNCTION main :  
2 DEC v6 40  
3 v1 := #30  
4 t1 := v1 / #5  
5 v2 := t1  
6 t2 := #9 - v2  
7 v2 := t2  
8 t3 := v2 * #4  
9 v3 := t3  
10 t4 := #2 * v3  
11 t5 := v1 - t4  
12 v4 := t5  
13 t6 := v4 * #4  
14 t7 := &v6 + t6  
15 t8 := *t7  
16 v5 := t8  
17 RETURN #0
```

输出：

在编译时，我们使用常量折叠技术，识别并计算常量表达式，以避免在运行时计算他们的值。通常，我们迭代式地遍历中间代码，基于复制传播及常量传播技术，用常量

代替一切识别为常量的变量或是表达式。对于以上的中间代码，优化后中间代码可以是这样的：

```
1 FUNCTION main :  
2 DEC v6 40  
3 v1 := #30  
4 t1 := #6  
5 v2 := #6  
6 t2 := #3  
7 v2 := #3  
8 t3 := #12  
9 v3 := #12  
10 t4 := #24  
11 t5 := #6  
12 v4 := #6  
13 t6 := #24  
14 t7 := &v6 + #24  
15 t8 := *t7  
16 v5 := t8  
17 RETURN #0
```

样例4（全局公共子表达式消除）：

输入：

```
1 FUNCTION main :  
2 READ t1  
3 v1 := t1  
4 READ t2  
5 v2 := t2  
6 t3 := v1 + #1  
7 v3 := t3  
8 t4 := v2 + #2  
9 v4 := t4  
10 t5 := v1 + v2  
11 v5 := t5  
12 t6 := v5 - v3  
13 v6 := t6  
14 t7 := v1 + v3  
15 v7 := t7  
16 t8 := v1 + v6  
17 v4 := t8  
18 v8 := #0  
19 LABEL label1 :  
20 IF v8 < v1 GOTO label2  
21 GOTO label3  
22 LABEL label2 :  
23 t9 := v8 + #1  
24 v8 := t9  
25 IF v1 < v2 GOTO label4  
26 GOTO label5  
27 LABEL label4 :  
28 t10 := v1 + v2  
29 v8 := t10  
30 t11 := v1 + #1  
31 v6 := t11  
32 GOTO label6  
33 LABEL label5 :  
34 t12 := v2 + v3  
35 v1 := t12
```

```
36 t13 := v1 + v2
37 v5 := t13
38 LABEL label6 :
39 GOTO label1
40 LABEL label3 :
41 RETURN #0
```

输出：

对于全局的公共子表达式的消除，通常我们采用可用表达式模式，分析到达某一程序点p的所有路径上，是否存在可用的表达式。全局公共子表达式消除后，中间代码可以是这样的：

```
1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := v1 + #1
7 v3 := t3
8 t4 := v2 + #2
9 v4 := t4
10 t5 := v1 + v2
11 v5 := t5
12 t6 := v5 - v3
13 v6 := t6
14 t7 := v1 + v3
15 v7 := t7
16 t8 := v1 + v6
17 v4 := t8
18 v8 := #0
19 LABEL label1 :
20 IF v8 < v1 GOTO label2
21 GOTO label3
22 LABEL label2 :
23 t9 := v8 + #1
24 v8 := t9
25 IF v1 < v2 GOTO label4
26 GOTO label5
27 LABEL label4 :
28 v8 := v5
29 t10 := v1 + #1
30 v6 := t10
31 GOTO label6
32 LABEL label5 :
33 t11 := v2 + v3
34 v1 := t11
35 t12 := v1 + v2
36 v5 := t12
37 LABEL label6 :
38 GOTO label1
39 LABEL label3 :
40 RETURN #0
```

样例5（常量传播）：

输入：

```

1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := v1 + #1
7 v3 := t3
8 v4 := #2
9 t4 := v3 * v4
10 v5 := t4
11 t5 := #2 + v3
12 v8 := t5
13 t6 := v3 * #2
14 v7 := t6
15 t7 := #2 * v3
16 t8 := t7 + #4
17 v9 := t8
18 IF v9 == v7 GOTO label2
19 LABEL label1 :
20 v3 := #3
21 t9 := v4 + #5
22 v5 := t9
23 t10 := #2 * v8
24 v1 := t10
25 t11 := v9 - v1
26 v2 := t11
27 GOTO label3
28 LABEL label2 :
29 v2 := v3
30 t12 := v5 + v2
31 v1 := t12
32 t13 := v7 * v3
33 v9 := t13
34 LABEL label3 :
35 t14 := v1 + v2
36 v8 := t14
37 v7 := v3
38 t15 := v5 - v2
39 v9 := t15
40 RETURN #0

```

输出：

对于全局常量传播的计算，现今较为通用的算法是稀疏条件常量传播算法，这是一种基于到达定值模式的方法，对于常量的传播过程进行抽象解释，合并不同分支上变量的赋值情况。在常量替换后，中间代码可以是这样的：

```

1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := v1 + #1
7 v3 := t3
8 v4 := #2
9 t4 := v3 * #2
10 v5 := t4
11 t5 := #2 + v3
12 v8 := t5

```

```

13 t6 := v3 * #2
14 v7 := t6
15 t7 := #2 * v3
16 t8 := t7 + #4
17 v9 := t8
18 v3 := #3
19 v5 := #7
20 t9 := #2 * v8
21 v1 := t9
22 v2 := #0
23 v8 := v1
24 v7 := v3
25 v9 := #7
26 RETURN #0

```

对于原有的分支跳转，在经过常量传播之后，发现只会执行true分支的语句，所以我们不分析false分支中对变量的重新赋值的情况，经过常量传播之后，中间代码变得更为简洁而高效。以上的中间代码中， $v9 \neq v7$ 恒成立，但是包含变量的表达式存在不同的情况，在简化时需注意（例如 $a+1$ 与 $a-1$ ， $a+1 > a-1$ 并不恒成立）。

样例6（全局无用代码消除）：

输入：

```

1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := v1 + #1
7 v3 := t3
8 t4 := v2 + #2
9 v4 := t4
10 t5 := v1 + v2
11 v5 := t5
12 v6 := #0
13 LABEL label1 :
14 IF v6 < v1 GOTO label2
15 GOTO label3
16 LABEL label2 :
17 v1 := v5
18 t6 := v1 - #1
19 v4 := t6
20 IF v3 < v4 GOTO label4
21 GOTO label5
22 LABEL label4 :
23 v3 := #4
24 v2 := v4
25 t7 := v1 + v2
26 v6 := t7
27 GOTO label6
28 LABEL label5 :
29 t8 := v3 + #3
30 v3 := t8
31 v6 := v1
32 t9 := v1 + #1
33 v6 := t9
34 LABEL label6 :

```

```
35 t10 := v6 + #3
36 v6 := t10
37 GOTO label1
38 LABEL label3 :
39 t11 := #2 * v1
40 v5 := t11
41 t12 := v2 + v5
42 v6 := t12
43 RETURN #0
```

输出：

对于一个变量在某个程序点上定义的值，若在之后的程序执行过程中未被使用，该定义语句是无用代码，可被削减。同时，对于不被执行的条件分支中的语句，也被认为是无用代码。对于无用代码消除，我们通常采取活跃变量分析的模式。生成的中间代码可以是这样的：

```
1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := v1 + #1
7 v3 := t3
8 t4 := v1 + v2
9 v5 := t4
10 v6 := #0
11 LABEL label1 :
12 IF v6 < v1 GOTO label2
13 GOTO label3
14 LABEL label2 :
15 v1 := v5
16 t5 := v1 - #1
17 v4 := t5
18 IF v3 < v4 GOTO label4
19 GOTO label5
20 LABEL label4 :
21 v3 := #4
22 v2 := v4
23 GOTO label6
24 LABEL label5 :
25 t6 := v3 + #3
26 v3 := t6
27 t7 := v1 + #1
28 v6 := t7
29 LABEL label6 :
30 t8 := v6 + #3
31 v6 := t8
32 GOTO label1
33 LABEL label3 :
34 t9 := #2 * v1
35 v5 := t9
36 t10 := v2 + v5
37 v6 := t10
38 RETURN #0
```

样例7（循环不变代码外提）：

输入：

```
1 FUNCTION main :  
2 READ t1  
3 v1 := t1  
4 READ t2  
5 v2 := t2  
6 t3 := v1 + #1  
7 v3 := t3  
8 t4 := v2 * #2  
9 v4 := t5  
10 t5 := v3 * v4  
11 v5 := t5  
12 t6 := v5 * v4  
13 t7 := v1 + t6  
14 v6 := t7  
15 t8 := v3 - v1  
16 v7 := t8  
17 LABEL label1 :  
18 IF v1 > v2 GOTO label2  
19 GOTO label3  
20 LABEL label2 :  
21 v1 := v7  
22 t9 := v7 * v4  
23 v2 := t9  
24 IF v3 > v4 GOTO label4  
25 GOTO label5  
26 LABEL label4 :  
27 v3 := #4  
28 t10 := v1 * #5  
29 v4 := t10  
30 GOTO label6  
31 LABEL label5 :  
32 t11 := v1 - #3  
33 v8 := t11  
34 LABEL label6 :  
35 GOTO label1  
36 LABEL label3 :  
37 t12 := #2 * v4  
38 v9 := t12  
39 RETURN #0
```

输出：

程序执行的过程中，大部分的执行时间都花在循环上，因此，研究人员针对循环设计了许多编译优化技术。对于循环内部不管执行了多少次，都能得到相同结果的表达式，我们可以将其移动到循环外部，以避免循环过程中反复执行该表达式。经过循环不变代码外提后，中间代码可以是这样的：

```
1 FUNCTION main :  
2 READ t1  
3 v1 := t1  
4 READ t2  
5 v2 := t2  
6 t3 := v1 + #1  
7 v3 := t3  
8 t4 := v2 * #2
```

```

 9 v4 := t4
10 t5 := v3 * v4
11 v5 := t5
12 t6 := v5 * v4
13 t7 := v1 + t6
14 v6 := t7
15 t8 := v3 - v1
16 v7 := t8
17 v1 := v7
18 LABEL label1 :
19 IF v1 > v2 GOTO label2
20 GOTO label3
21 LABEL label2 :
22 t9 := v7 * v4
23 v2 := t9
24 IF v3 > v4 GOTO label4
25 GOTO label5
26 LABEL label4 :
27 v3 := #4
28 t10 := v1 * #5
29 v4 := t10
30 GOTO label6
31 LABEL label5 :
32 t11 := v1 - #3
33 v8 := t11
34 LABEL label6 :
35 GOTO label1
36 LABEL label3 :
37 t12 := #2 * v4
38 v9 := t12
39 RETURN #0

```

样例8（强度削减）：

输入：

```

1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := #4 * v1
7 v3 := t3
8 t4 := #4 * v2
9 v4 := t4
10 t5 := v1 - #1
11 v5 := t5
12 t6 := v2 * #2
13 v6 := t6
14 t7 := v1 + v2
15 v7 := t7
16 t8 := v3 * v5
17 v8 := t8
18 t9 := v7 - v4
19 v9 := t9
20 LABEL label1 :
21 IF v3 < v9 GOTO label2
22 GOTO label3
23 LABEL label2 :
24 t10 := v1 + #1
25 v1 := t10
26 t11 := #4 * v1

```

```

27 v3 := t11
28 t12 := v3 * v4
29 v2 := t12
30 LABEL label4 :
31 IF v4 > v5 GOTO label5
32 GOTO label6
33 LABEL label5 :
34 t13 := v2 - #1
35 v2 := t13
36 t14 := #4 * v2
37 v4 := t14
38 GOTO label4
39 LABEL label6 :
40 GOTO label1
41 LABEL label3 :
42 RETURN #0

```

输出：

对于循环内部的昂贵操作，我们可以对其做一些语义相同的转化，将“昂贵”的操作转化为相对便宜的操作。我们运用强度削减，对其中可以简化的乘法操作转化为加法或减法操作，生成的中间代码可以是这样的：

```

1 FUNCTION main :
2 READ t1
3 v1 := t1
4 READ t2
5 v2 := t2
6 t3 := #4 * v1
7 v3 := t3
8 t4 := #4 * v2
9 v4 := t4
10 t5 := v1 - #1
11 v5 := t5
12 t6 := v2 * #2
13 v6 := t6
14 t7 := v1 + v2
15 v7 := t7
16 t8 := v3 * v5
17 v8 := t8
18 t9 := v7 - v4
19 v9 := t9
20 LABEL label1 :
21 IF v3 < v9 GOTO label2
22 GOTO label3
23 LABEL label2 :
24 t10 := v1 + #1
25 v1 := t10
26 t11 := v3 + #4
27 v3 := t11
28 t12 := v3 * v4
29 v2 := t12
30 LABEL label4 :
31 IF v4 > v5 GOTO label5
32 GOTO label6
33 LABEL label5 :
34 t13 := v2 - #1
35 v2 := t13
36 t14 := t14 - #4
37 v4 := t14
38 GOTO label4

```

```
39 LABEL label6 :  
40 GOTO label1  
41 LABEL label3 :  
42 RETURN #0
```

6.2 实验指导

经过词法分析、语法分析、语义分析后，编译器前端的工作告一段落。中端将源代码翻译成中间代码表示形式，机器无关优化基于中间代码进行代码优化。出于安全性或编程习惯的考量，源代码中通常有大量冗余代码。程序的冗余通常导致编译器执行效率低下，时空开销高昂，并且最终导致生成目标代码运行效率低下等问题。区别于与目标机器相关优化，机器无关优化不考虑目标机器的寄存器与机器指令。基于中间代码的优化是不需考虑源语言与目标语言存在的差异，独立于前端和后端进行优化的全部过程。有哪些程序片段需要被优化，应当运用哪些优化方法，应当以什么顺序进行代码优化？这些问题决定了优化时的时空开销，及最终程序优化的效果。

由于程序语言的复杂性、程序性质的多样性，针对程序的分析往往具有相当大的难度。根据 Rice 定理：对于程序行为的任何非平凡属性，都不存在可以检查该属性的通用算法。因此，没有一种优化的方式能够宣称，优化的结果到了最佳，这一消极的结论在另一方面有着积极的影响：总能够提出更好的优化方式，使得代码更加简洁与高效。

6.2.1 中间代码优化概述

在程序执行的任何给定点，当前程序使用数据在内存中的存储位置与内存位置的内容构成了程序状态。程序命令的顺序执行，本质上是对内存中存储的数据的定义与使用操作，操作的执行改变了程序状态。比如，我们声明了一个变量x并定义，就会开辟一段内存来存储x所代表的值。

中间代码优化的基础在于跟踪并分析程序状态的改变。比如，在某个程序点上，当前的变量是否具有唯一的常量，如果是，那么我们就可以将这个变量替换成一个常量。又或许，在某个程序点上，一个变量存储的值是否会在被使用之前就被覆盖掉，如果是，我们就不需要在内存或是寄存器里存储这个值。

一系列地连续重写IR以消除效率低下和无法轻易转换为机器代码的代码片段的方法或函数构成了编译器的中间代码优化模块。这些方法或函数通常被称为**趟(Pass)**。机

器无关优化模块排列趟的执行顺序，构成编译器的**优化管道(Pipeline)**。在某些编译器中，IR格式在整个优化管道中保持固定，在其他编译器中，格式在某些趟执行完毕后会发生改变。对于格式固定的优化管道，趟的顺序是相对灵活的，可以运行大量优化序列，并不会导致编译错误或是编译器崩溃。

根据处理粒度的不同，优化通常分为局部代码优化（基本块内部）、全局代码优化（多个基本块）和过程间代码优化（跨越函数边界）三种。这三种优化在优化管道中按顺序执行。

根据冗余原因的不同，优化通常分为公共子表达式削减、常量传播优化、循环相关优化、死代码消除、别名相关优化、内存相关优化等。

根据不同的优化需求，优化通常使用不同的数据流分析模式进行分析。常用的数据流分析模式有到达定值（针对循环削减、常量传播等），可用表达式（针对全局公共子表达式等），活跃变量分析（针对死代码削除等）。

优化管道中趟的顺序由编译器开发人员设计，合理的优化顺序能够使优化模块在合理的时空开销下获得更好的优化效果（如全局公共子表达式应在复制传播之前），而不合理的顺序使得优化管道需要反复执行同一个优化方法。因此，编译器研究中一个重要的研究课题是构造更好的优化管道。

在实验五中，你应当生成与实验三格式相同但是更加简洁高效的中间代码。而你设计的优化管道中趟的构造与执行顺序、生成中间代码的语义一致性、代码运行时间、执行操作次数等将作为评估指标。

6.2.2 基本块与控制流图

实验五中，我们基于实验三生成的中间代码进行**优化管道(optimization pipeline)**的设计。你可能会发现，实验三中生成的中间代码，其中包含了大量的跳转语句，程序的执行没有明显的顺序，使得你很难分辨代码的执行逻辑。

为了更好地描述程序中的值被定义和使用的顺序，我们可以使用控制流图的形式，对程序代码进行划分。控制流图是一个有向图，其中节点表示一个基本块，有向边表示基本块之间的跳转。有向图中节点的拓扑排序代表了程序的执行顺序。

基本块(basic block, BB): 程序执行过程中，只能从基本块的第一个指令进入该块，从最后一个指令离开该块。每次程序执行过程中访问基本块时，必须按顺序从头到尾执行其中每一条指令的代码片段。

对于基本块内部的每一条指令，都代表了一种程序操作或行为，在基本块中代码的执行过程中，依次执行这些行为。通常，我们使用程序状态图进行程序内部行为的抽象描述，状态图记录每一个指令所代表的行为，会对程序的输出造成的影响。基本块中代码按照顺序改变程序状态这种受限的形式，使得基本块非常易于分析。

控制流图(control-flow graph, CFG): 程序执行过程中所有可能路径的表示。基本块与跳转关系，代表了程序执行时程序代码遍历的顺序。通常，研究者根据代码执行的顺序，及代码执行时必然成立的性质，分析程序的状态，从而根据程序状态进行优化与静态检查。

以下这段代码贯穿局部优化与全局优化实例，其构造的控制流图如图2所示。

```
1 a = read();          21 if(a > b){  
2 b = read();          22   f = f + d;  
3 c = read();          23   h = a + 3 * y;  
4 d = a + b;          24 }else{  
5 e = c * b;          25   while(j > 0){  
6 f = a + b;          26     j = j - 1;  
7 f = 5;              27     g = e / (f - 1);  
8 g = e + d;          28     h = 4 * j;  
9 h = c * f;          29     if(y < d){  
10 x = b - 3;         30       g = f + h;  
11 y = 2;             31       x = a * c;  
12 a = x - y;         32     }  
13 b = e - h;         33   }  
14 i = 0;             34 }  
15 j = 10;            35 }  
16 while(i < 10){      36 write(x);  
17   i = i + 1;         37 write(h);  
18   d = b - a * c;    38 write(g);  
19   e = 4 * i;          20   g = x + 4;
```

图1 源代码示例

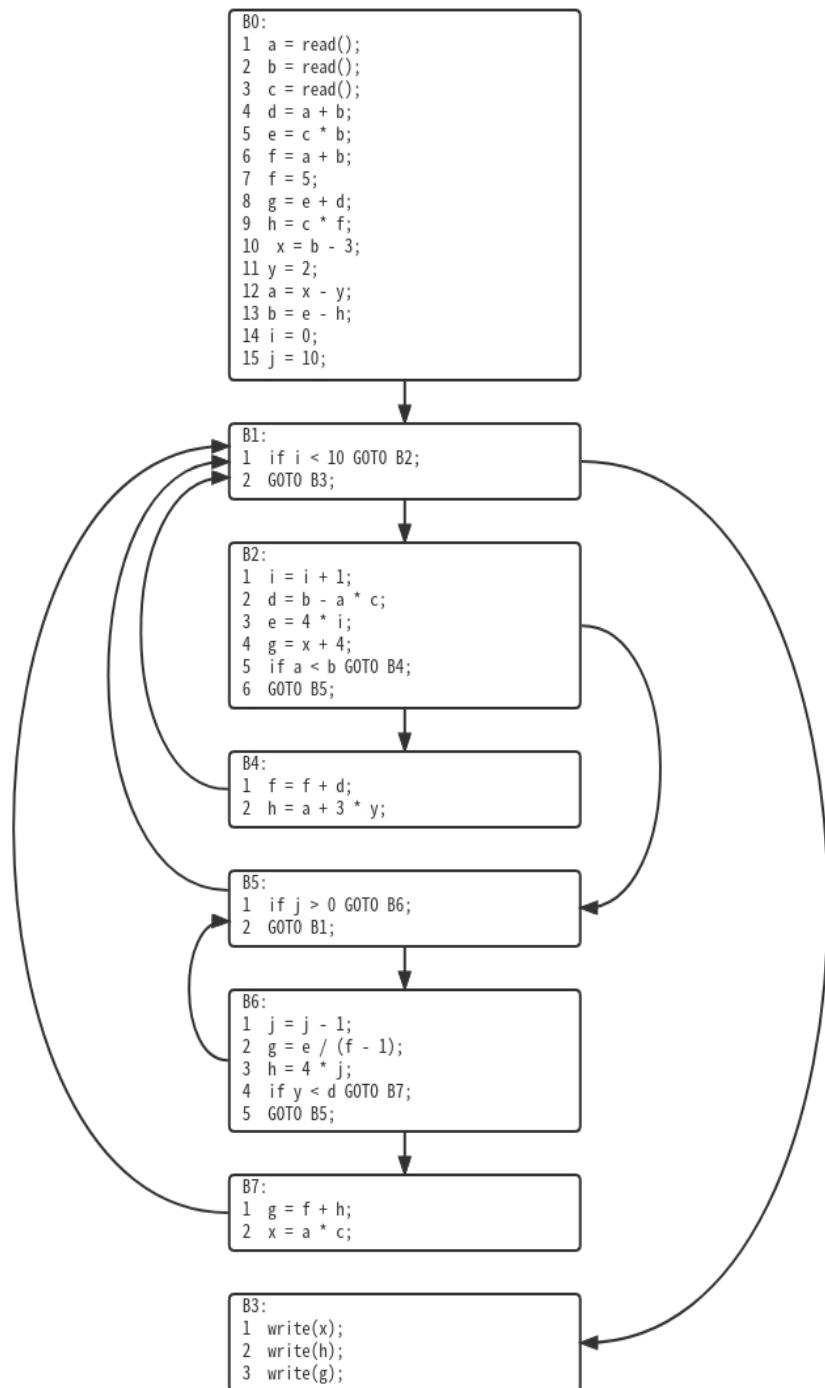


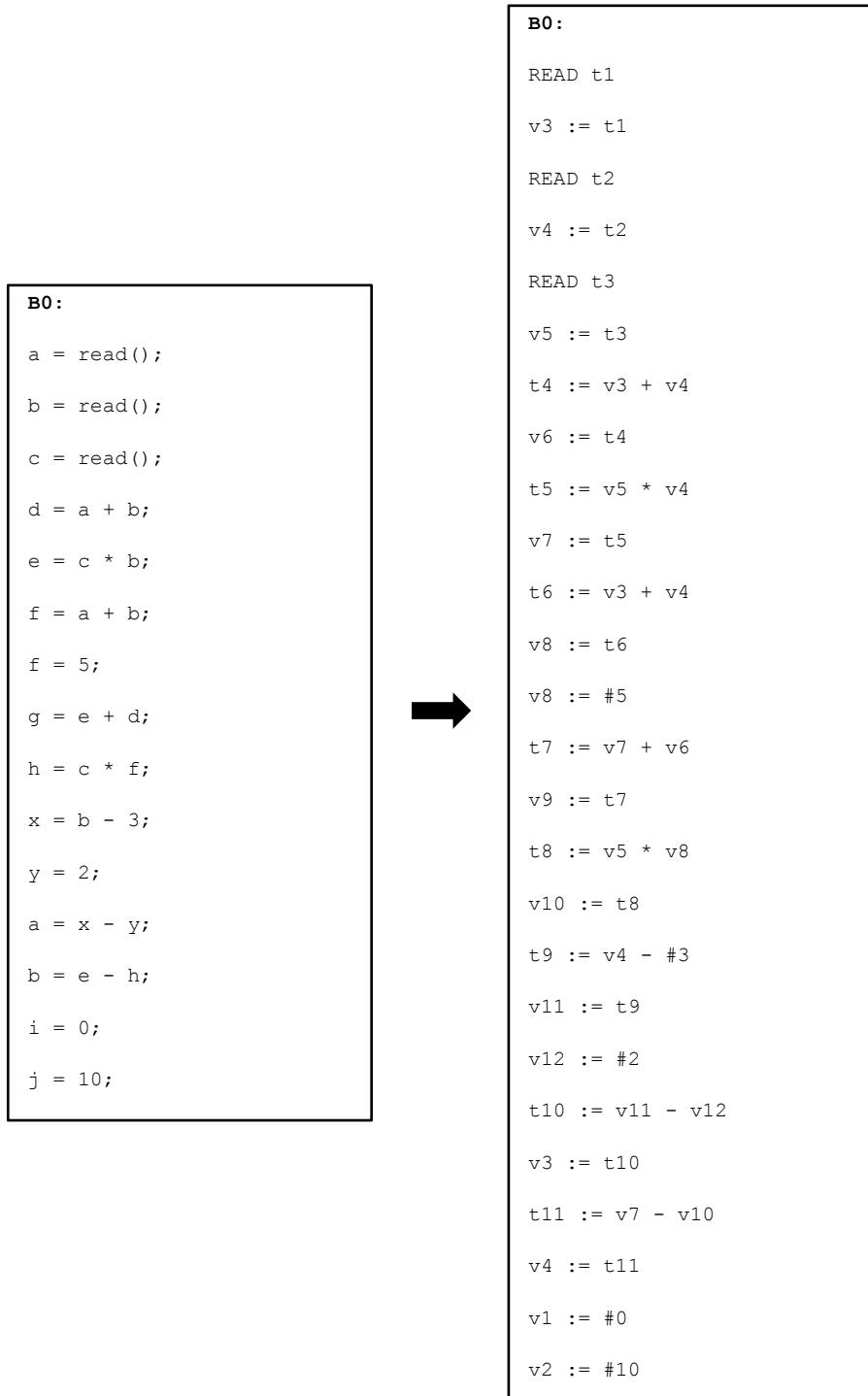
图2 控制流图

在稍后的部分中，我们将介绍基于控制流图的数据流分析方式，全局优化通常都会采用数据流分析技术进行实现。在这之前，我们先对每一个基本块内部进行局部优化。

6.2.3 局部优化

相对于全局优化而言，对于基本块内部的局部优化较为简单。大量局部优化技术需要将基本块内部的指令转化为一个**有向无环图(DAG)**。通过生成的有向无环图，我们能够轻易地揭示基本块内部的结构，可视化基本块之间的值流。

我们以生成的程序控制流图中的基本块B0为例，首先将源代码翻译为中间代码，然后生成如图3所示的有向无环图：



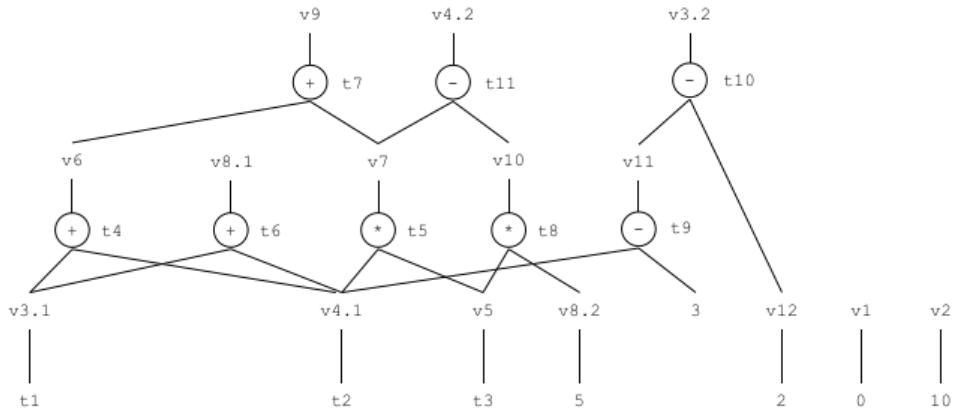


图3 有向无环图

有向无环图中的节点分为三类：叶子节点、根节点和普通节点。叶子节点表示常量或是定值；根节点表示当前基本块内部赋值但未使用的变量；节点标号中的运算符及子节点构成了赋值表达式，表达式求值的结果作为一个定值，标记在节点右侧；对于只有一个子节点的节点，其与子节点构成的节点对，表示该子节点代表的定值赋值给变量的过程。

基于构造的有向无环图，我们可以进行如下的优化：

1) **公共子表达式消除(common subexpression elimination)**: 如果表达式E在某次出现之前已经被计算过，并且表达式中的值在计算后一直没被修改过，那么E的出现被称为是公共子表达式。

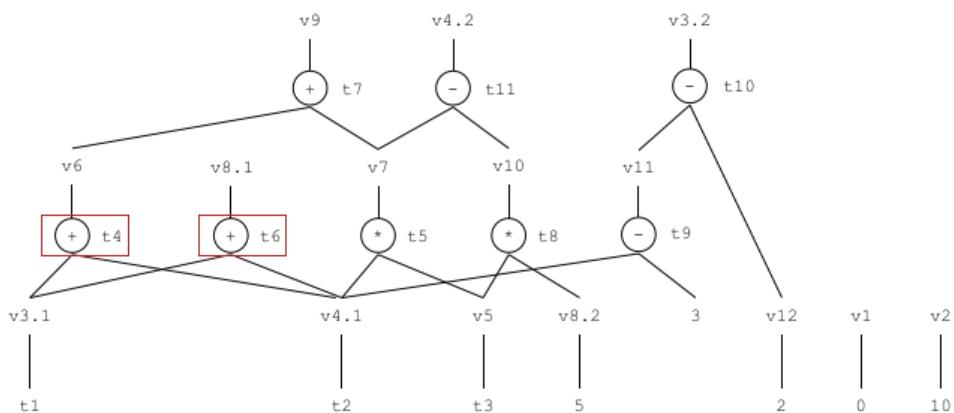


图4 公共子表达式示例

通过观察图4我们可以发现，有向无环图中标记为t₄与t₆的两个节点，节点标号的运算符与所有子节点均相同，因此，对于t₄的求值结果可以直接用于对t₆的求值中。因此可以使用t₄代替t₆，完成对子表达式的消除。

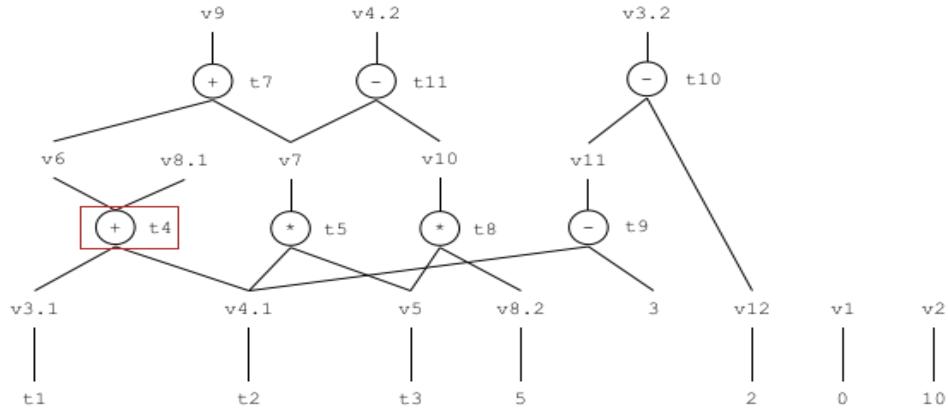


图5 公共子表达式消除

2)无用代码消除(dead code elimination): 你可能注意到，在基本块Bo内部，有一些变量在定义后从来没有被使用，或两次定义之间没有对该变量的使用语句。对于永远不会被使用的定义，为之开辟内存、进行计算是没有必要的。

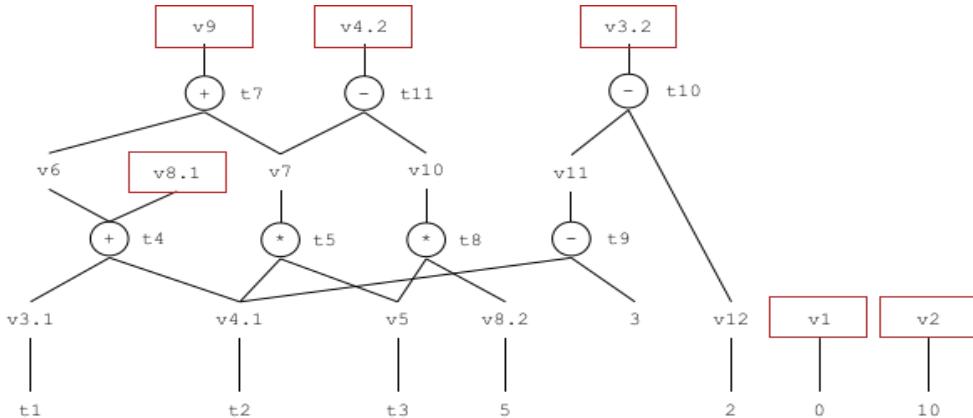


图6 无用代码示例

如图6所示，有向无环图中的根节点表示该变量在定义后未被使用，或两次定义之间没有使用语句。图中共有v1、v2、v3.2、v4.2、v8.1、v9，然而只有v8.1在基本块的出口不是活跃的（在后续执行过程中未被使用），因此可消除v8.1及其对应的赋值语句，转化后的有向无环图如图7所示。

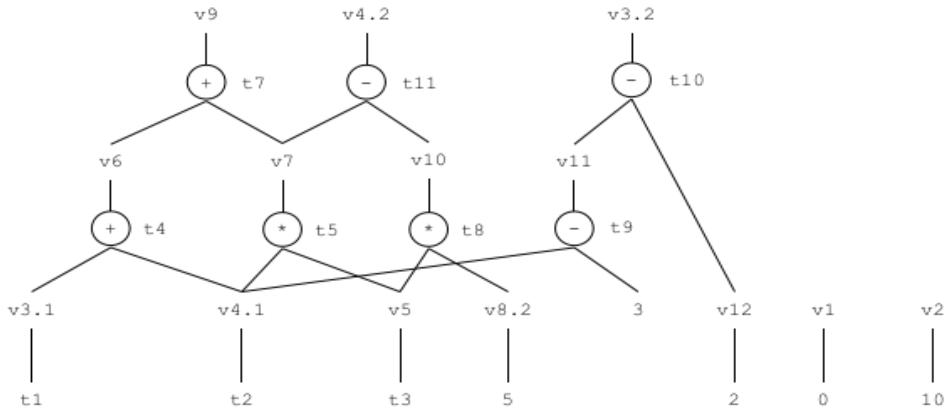


图7 无用代码消除

3)常量折叠(constant folding): 我们可以发现,对于程序内部如 $f = 5$, $y = 2$, $i = 0$, $j = 10$ 等将变量定义为一个常量的赋值语句,在该变量被再次定义之前,可将变量替换为一个常量,从而简化计算的过程。

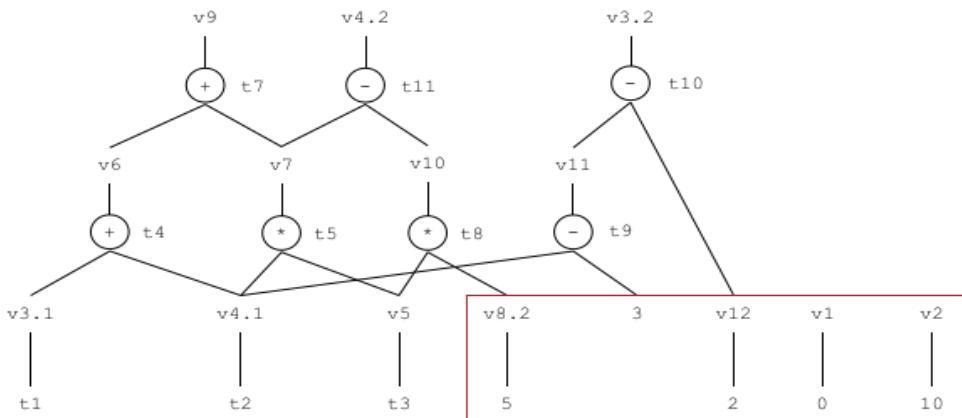


图8 被赋值为常量的变量示例

从图8中的叶子节点,我们可看出,被定义为常量的变量有 $v1$ 、 $v2$ 、 $v8.2$ 、 $v12$ 。在使用的过程中我们能够使用常量值代替其参与运算。并且,我们可以基于代数恒等式计算,尽可能地简化与合并常量值。

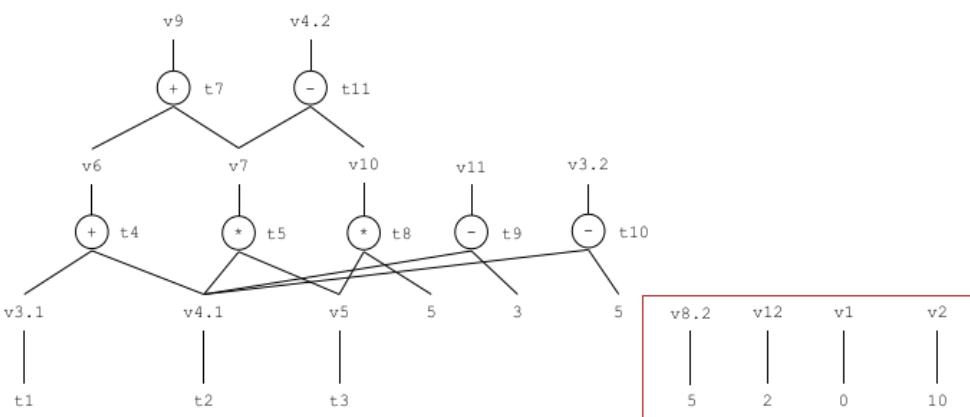


图9 常量折叠

基于有向无环图进行常量折叠之后，有向无环图的结构发生改变，可能导致无用代码的暴露。同时，对于生成的有向无环图进行代数恒等式替换，能够发掘代码内部的公共子表达式，如图10所示。

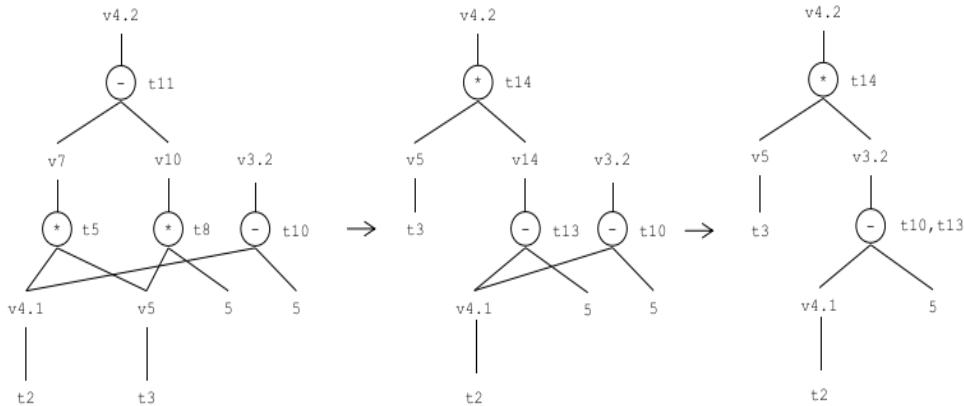


图10 代数恒等式替换

在图10中，经过代数恒等式的转化，能够发掘出 $v4.1 - 5$ 这一公共子表达式，并简化生成的中间代码。同时，若 $v7$ 在后续代码中未被使用，则可进行无用代码消除，去除对 $v7$ 的赋值语句。

经过以上的优化策略，我们将优化后的有向无环图还原为中间代码形式。我们可以适当地改变中间代码顺序，提升最终的优化效果。最终生成的有向无环图与中间代码如下所示：

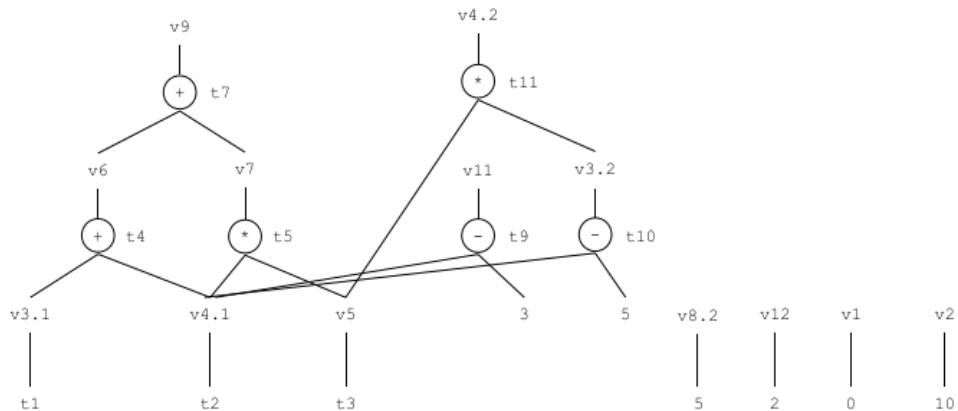
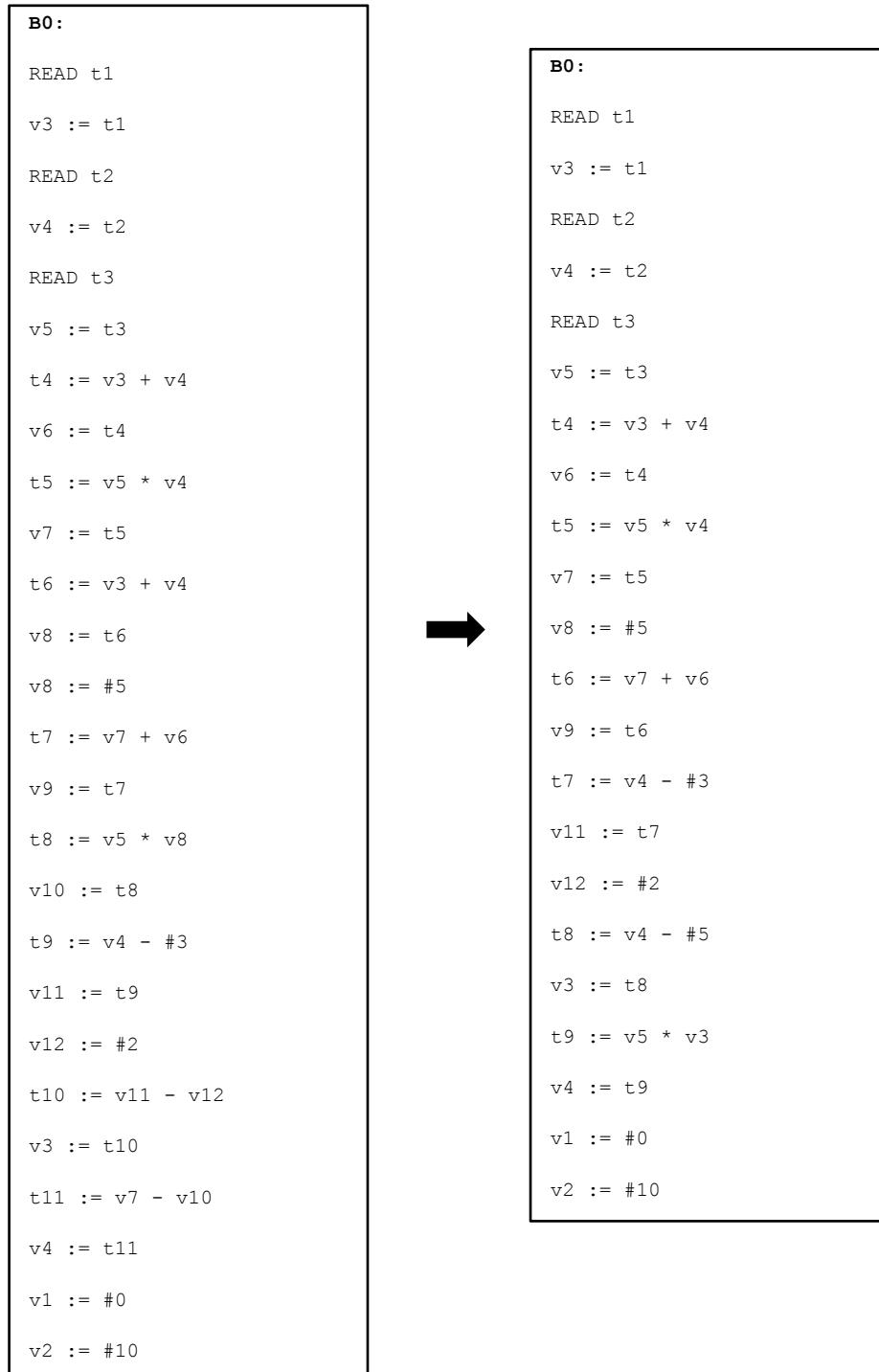


图11 优化后的有向无环图

优化后的中间代码如下所示：



完成了单个基本块的优化工作，我们将视野放大到整个函数，进行多个基本块上的优化工作。

6.2.4 数据流分析概述

在局部优化中，你可能隐约地感受到，在无用代码消除中使用的 defined-use 对，揭示了一个变量中存储的数据沿着程序执行路径流动的现象，只不过，在基本块中数据

的流动是线性的，而全局的情况下，因存在循环、函数调用等情况，数据的流动变得复杂。比如，全局公共子表达式消除就需要确定在程序的任何可能执行路径上，是否都存在内容相同且值未改变的表达式。

为了在复杂的控制流中发掘这种数据流动的关系，编译器的设计者将**数据流分析**(data-flow analysis)技术引入到全局优化工作之中。基于前文所述，我们可以将程序执行的过程看作是一系列程序状态的转换，而针对特定的问题，我们只需要进行**抽象解释**(abstract interpretation)从程序状态中抽取对应的信息，来解决特定的数据流分析问题。比如，对于一个变量在某个程序点上是否为定值的问题(到达定值问题)，只需要抽取该变量赋值的相关语句。如图 12 所示，我们想知道，当 x 在基本块 B5 中被输出时，是否是一个定值。我们首先抽取所有与 x 赋值相关的语句，然后使用数据流分析方法分析在所有可能的执行路径上， x 的赋值结果。通过对数据流的分析能够得知，当执行 $\text{write}(x)$ 语句时， x 的值为定值 b 。

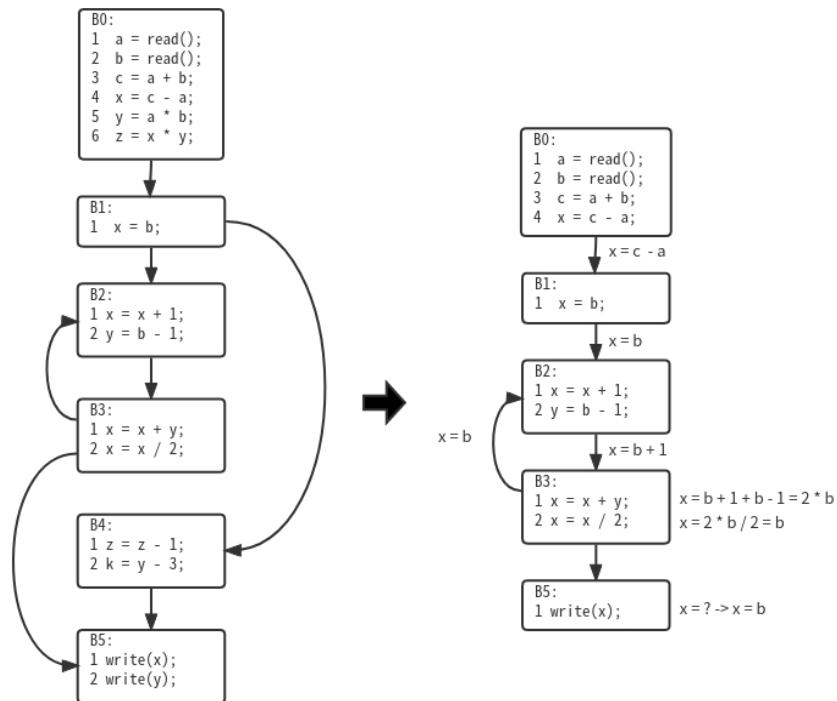


图 12 抽象解释示例

因现代软件代码规模的急剧增长，使用数据流分析技术进行程序状态的跟踪与计算时，常常需要进行精度与时空成本之间的博弈，在追求较小的时间与空间成本进行程序

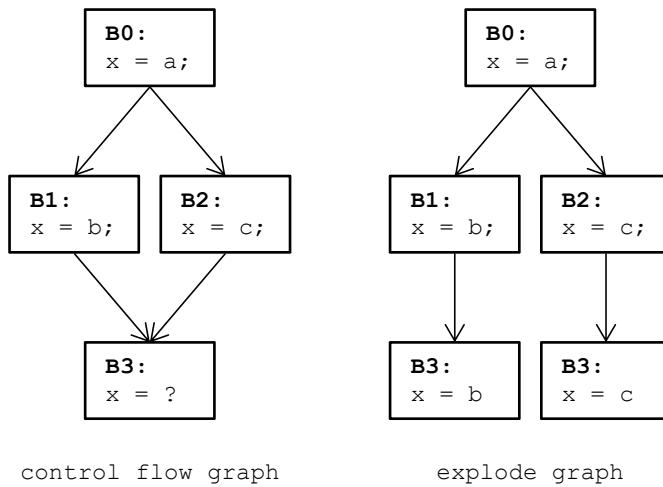
状态的分析时往往也意味着较低的分析精度。我们用对不同情况的敏感性来描述我们采取的分析方法的预期精度。

1) **流敏感与流不敏感(flow-sensitive/insensitive)**: 程序内部数据随着程序的执行顺序流动，流敏感的分析根据程序的执行顺序跟踪程序状态的变化，而流不敏感的分析通过代码扫描报告所有可能出现的情况。

```
x = 1;  
x = 2;
```

对于以上的两条语句，使用流敏感的分析方法，分析结果会告诉我们：在执行第一条语句后， x 被赋值为1，在执行第二条语句之后， x 被赋值为2。使用流不敏感的分析方法，分析结果会告诉我们： x 的值可能为1或2。

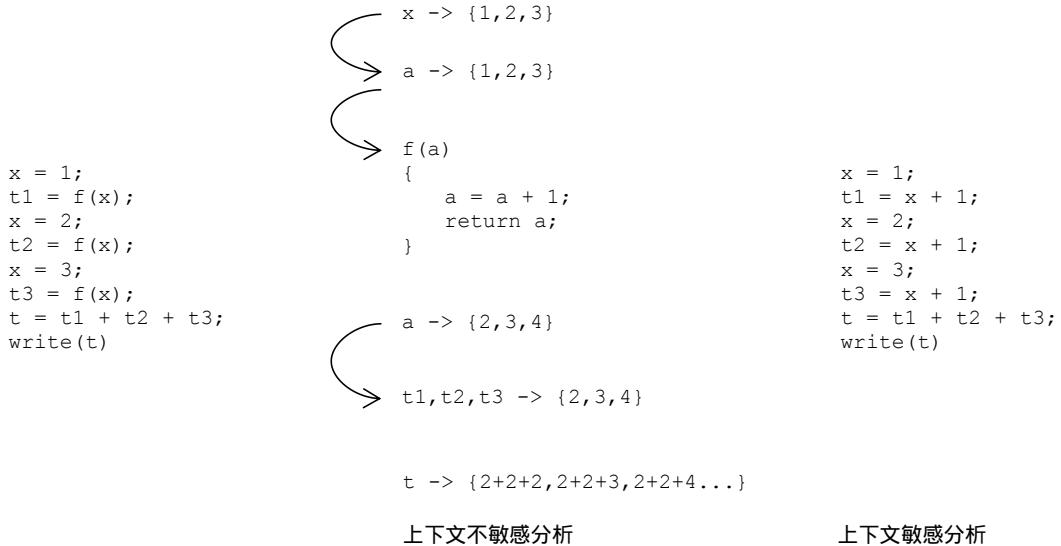
2) **路径敏感与路径不敏感(path-sensitive/insensitive)**: 路径敏感分析与路径不敏感分析的区别在于：路径敏感分析将构造的程序控制流图(control-flow graph)扩展为扩展图(exploded graph)，跟踪程序内部所有可能路径，分析程序状态变化，而路径不敏感分析通常只基于数据流图进行分析。



当使用路径不敏感的分析时，分析结果会告诉我们： x 的值可能为b或c，而基于扩展图进行分析时，能够直观展现在不同的执行路径上对 x 的值的改变。

虽然路径敏感的分析精度更高，但是可能存在路径爆炸(path explosion)的问题。例如，对于循环的分析，每次执行循环都会多出大量的可执行路径，极大地增加了分析成本。

3) 上下文敏感与上下文不敏感(context-sensitive/insensitive): 上下文敏感性与函数调用相关。上下文敏感的分析方法关注在函数调用时调用点的程序状态，而上下文不敏感的分析方法不考虑函数调用点的信息。



上下文不敏感分析中，输入值存在三种可能性：1, 2, 3，从而对于函数的返回值也有三种可能：2, 3, 4，于是t的最终取值可能为6-12中的任何值。对于上下文敏感分析，使用函数内联的形式进行体现，能够精准得出t值为 $2+3+4=9$ 。

程序状态改变函数:

我们把每个语句s之前和之后的程序状态分别记为IN[s]和OUT[s]，对于程序状态的改变存在两种情况：执行命令对程序状态的改变和控制流带来的程序状态的改变。

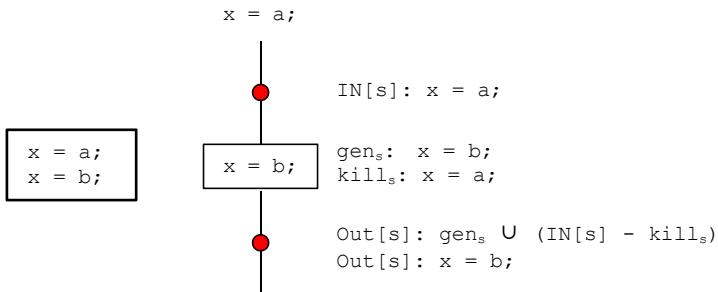
传递函数:

$$OUT[s] = f(IN[s])$$

传递函数描述了在进行数据流分析的过程中，模拟执行一条语句并且分析其对程序状态产生的变化。通过对传递函数的设计，能够描述不同的抽象解释的情况下，我们所关注的程序状态的改变情况，如，当我们想知道变量的赋值变化的情况，我们构造的传递函数可以是这样的：

$$\text{传递函数: } f_s = \text{gen}_s \cup (\text{IN}[s] - \text{kill}_s)$$

其中， gen_s 表示当前语句生成的赋值关系（define）， kill_s 表示因当前语句而失效的赋值关系。



控制流约束函数：对于不同的程序抽象与敏感性，我们需要设定不同的控制流约束函数，合并不同的分支路径上的程序状态。例如，当我们进行无用代码消除时，只需要有一条路径能够执行到基本块，即可认为基本块内语句不是无用代码。而当进行指针指向分析时，对于空指针解引用的检查，需要保证所有的可执行路径，分析的指针均已被赋值。

后文所述的数据流分析算法就是基于设计的传递函数与控制流约束函数，迭代地计算程序状态，直到程序状态不再改变为止。

May分析与Must分析：

May分析用于分析在某一程序点上所有可能存在的程序状态。例如，对于到达定值分析，我们想要知道在某一程序点p上变量的赋值情况，那么，我们使用may分析，对所有包含程序点p的路径进行分析。

Must分析用于分析在某一程序点上一定存在的程序状态。例如，对于可用表达式分析，我们想要知道在某一程序点p上，已被求值的表达式的情况，那么，对于经过程点p上的所有路径，该表达式均已被求值，且构成表达式的变量均未被赋值，那么在该程序点上表达式才是可用的。

分析准确性：

你或许想知道，在进行数据流分析时，是否能获得我们想要的结果。如果我们设计了一个优化算法，改变了代码的语义，消除了不应消除的语句，会直接改变程序执行的结果，使得执行不符合预期，这样就违反了我们优化的初衷。如果我们设计的优化算法，对程序没有进行任何优化，那么，虽然程序语义没有改变，我们的优化算法也失去了意义。

那么我们选择的分析方式能够精确地去除冗余代码，（或尽可能去除冗余代码），降低程序开销，而不改变程序语义吗？

在后文所述的数据流分析模式中，我们使用**迭代算法(iterative algorithm)**或**工作表算法(worklist algorithm)**计算程序状态的变化。通过以下三种分析方式的比较，我们简单地衡量所使用的算法的准确性：

IDEAL solution = 合并所有可执行的路径上的程序状态

MOP(meet over all paths) = 合并所有路径上的程序状态

MFP(maximal fixedpoint) = 迭代算法的结果

我们定义优化结果的五种状态：

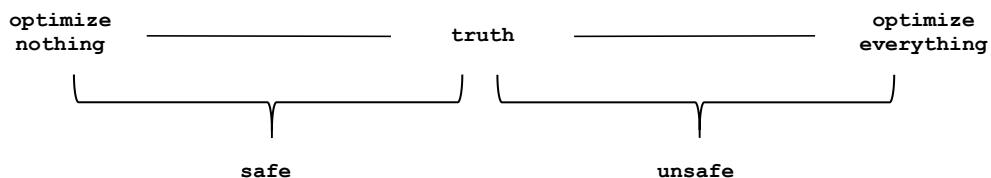
safe optimization: 优化后不改变程序语义

unsafe optimization: 优化后改变程序语义

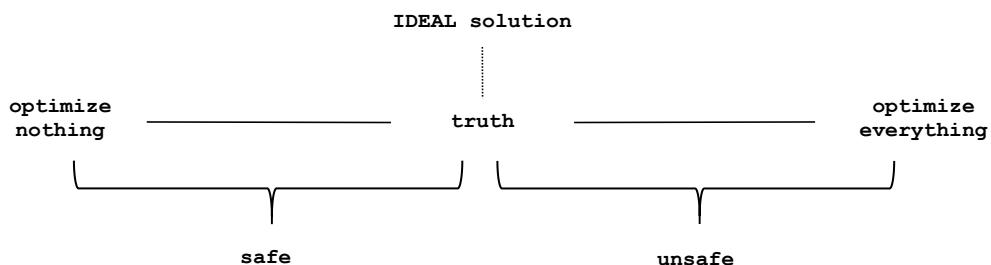
truth: 优化掉所有冗余代码，且语义相同的代码开销最小

optimize nothing: 不优化任何代码

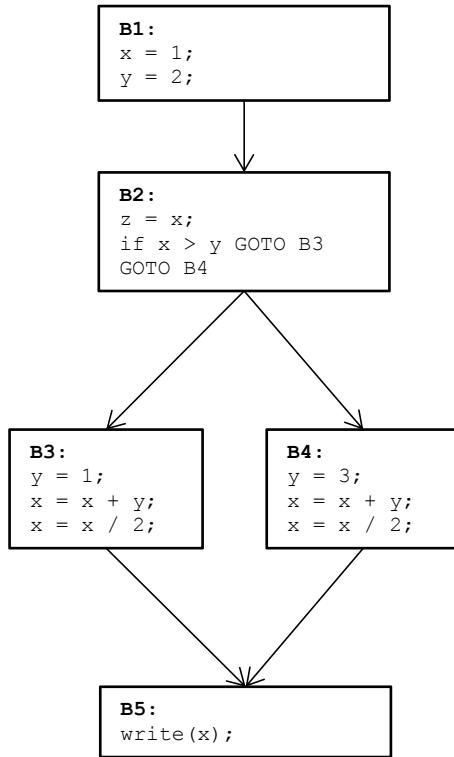
optimize everything: 优化任何代码



显而易见，IDEAL solution是理想化的分析解，是对实际运行过程中的可执行路径上的程序状态的分析，其分析结果为Truth。

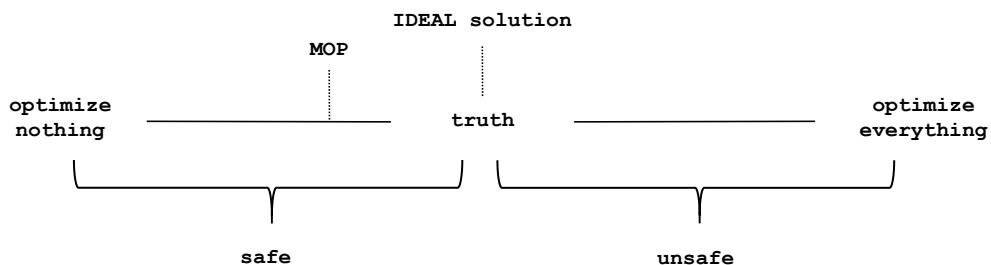


那么，我们首先把MOP的结果与IDEAL solution比较。下图使用公共子表达式消除举例，使用must analysis对其进行分析。

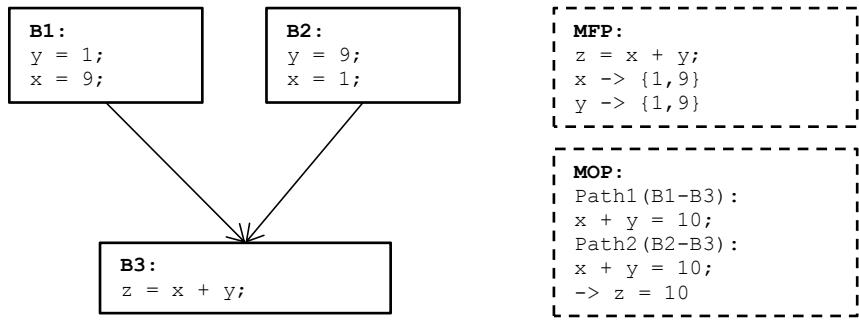


对于上图所示的这段代码，在程序实际执行的过程中，B2-B3-B5这条路径是不可执行的。但是，若我们未对代码进行常量传播分析之前，分析基本块B5中的表达式 $x-y$ 是否已被计算过，因不可执行路径上不存在对该表达式的计算，因此不认为 $x-y$ 在基本块B5中可以作为公共子表达式消除。

因此，IDEAL solution中应当消除 $x-y$ 这一公共子表达式，而相比IDEAL solution而言，MOP分析了更多路径上的程序信息，使得 $x-y$ 没有被消除，因此MOP的分析结果比IDEAL solution更为保守。

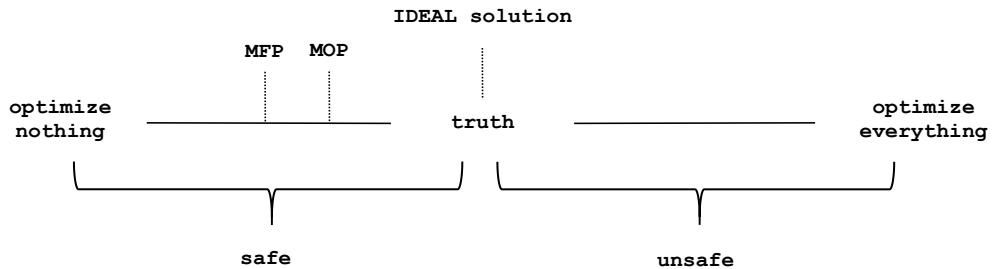


然后，对于MOP与MFP的比较，我们使用一个常量传播的例子用于解释其中的差别：



对于上图所示的这段代码，MOP跟踪不同的执行路径并进行计算，对于两条路径，`z`均为一定值10，因此可以使用常量进行替换，而MFP在基本块的入口使用控制流约束函数，合并不同前驱的程序状态，当判断`z`是否是常量时，`x`的可能取值有两种，`y`的可能取值也有两种，那么当进行常量传播分析时，不能将`z`转化为常量值10。

使用前文所述的敏感性来表示两种方法的不同，MOP是路径敏感的分析，而MFP是路径不敏感的，因此使用MFP的方式进行分析，其结果比MOP的更为保守。



你可能会想，我们分析的目的，就是一定要达到IDEAL solution。然而，任何路径敏感的分析，都无法避免地会遇到上文所述的路径爆炸问题。同时，进行路径敏感的分析，需要记录所有可能路径上程序状态，分析时的时空成本也极大，不适于在大规模的程序上分析程序状态。

我们将要学习的数据流分析模式是过近似(over-approximate)的、追求保守解(sound static analysis)的，这代表分析结果得出的优化策略，在优化之后不会改变程序的语义的同时，对于我们的分析结果，也不能确保分析的结果是最好的。

因此，我们使用的分析，无法保证所有可被消除的代码都已被消除。实际上，根据莱斯定理(Rice's Theorem)，不存在通用、高效的算法，使得优化达到最好的效果，没有任何一种优化的算法，是最好的。

在实验五中，你需要针对不同的优化目的，对程序进行抽象，构造程序状态转变的函数，描述并且记录程序状态的变化情况，基于获取的状态信息，完成优化工作。

6.2.5 数据流分析模式（到达定值）

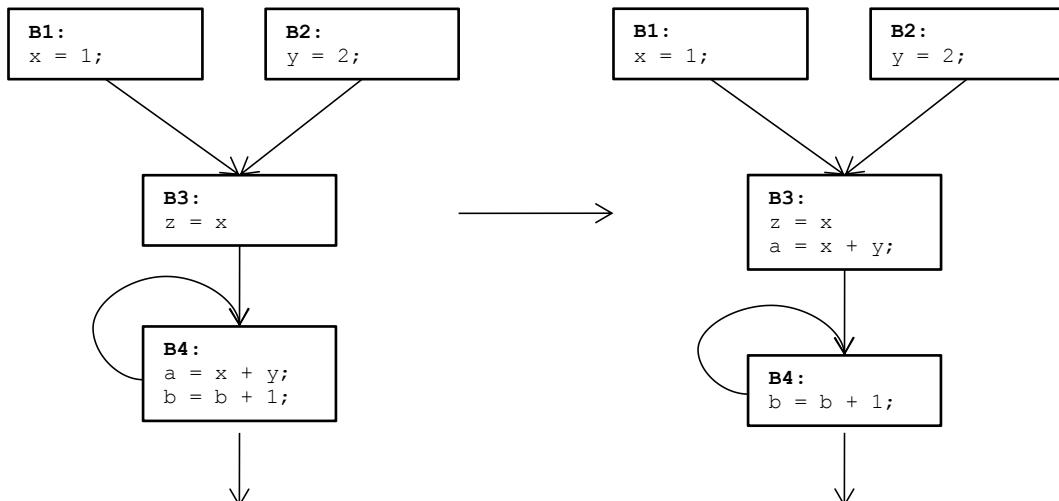
在数据流分析中，我们通常需要了解，数据在哪里被定义，在哪里被使用。在使用时，当前使用的变量是否已经被定义？使用未被定义的变量会诱发“未定义的引用”问题。当前使用的变量是否是一个定值？若是，那么在编译时就可以用一个定值代替这个变量。是否存在定义而从未被使用的情况？若有，该定义是“无用”的。在局部优化中，我们已经在基本块内部探究了变量的define-use关系，而数据流分析则试图揭示多个基本块乃至多个函数之间变量的define-use关系。

到达定值(reaching definition)是最基础的数据流分析模式之一，描述了在程序内部的每个程序点上，变量可能的赋值情况，是与程序内部的变量的define-use关系最相关且最简单的分析模式。

到达定值分析的主要用途有以下几项：

循环不变代码外提(loop-invariant code motion, LICM)：

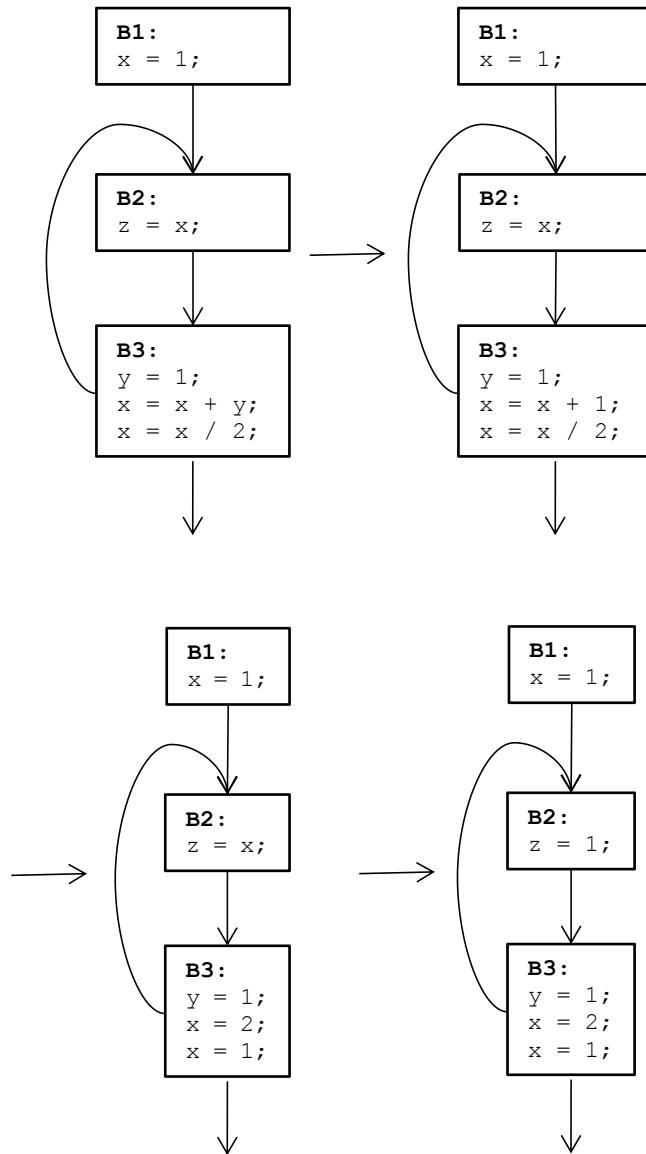
对于循环内部的赋值语句，若构成赋值表达式的变量均在循环外部定义，则可以将该语句移动到循环外侧，降低执行时的开销。



常量折叠 (constant folding) :

我们可以迭代式地将程序内部可转变为常量的变量转化为常量，降低执行的开销。

对于下面这段代码中的变量x, y, z，使用到达定值模式和数据流分析算法进行分析，可以将其中的所有变量都替换为常量。



到达定值模式应用：

我们首先要选择，我们是应当使用may分析还是must分析进行程序状态的分析。

对于到达定值问题，为了分析在某一程序点p上的变量v，对于所有经过程点p的路径，变量v的值是否恒为一常量，即，我们需先获取v在程序点p上所有可能的值的情况，再判断所有可能值是否均为一相同定值，为此，我们使用may分析的方式进行程序状态的分析。

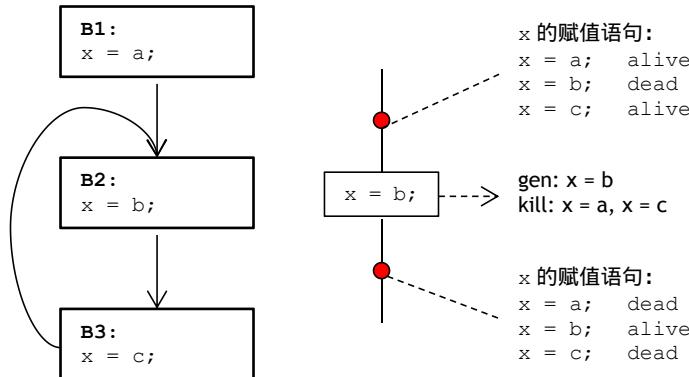
然后，我们要构造针对到达定值问题的传递函数与控制流约束函数。

前文我们简要地介绍了程序状态的传递函数，接下来我们深入地剖析一下传递函数各部分的语义，以便于我们对传递函数的理解。

传递函数用于描述程序内部的状态变化，那么，对于到达定值问题，我们只需关注其中那些针对变量的赋值语句。程序内部通常存在多条对同一变量v赋值的语句，但是在某一条对变量v的赋值语句 $s: v = d$ 执行完毕之后，v的值只可能是“定值”d。于是，对于变量v，其状态转换的传递函数定义为：

$$f_s = \text{gen}_s \cup (\text{In}[s] - \text{kill}_s)$$

我们将该传递函数运用到如下的程序中，描述程序状态的变化：



程序内部对变量x的赋值语句共有3条： $x = a$, $x = b$, $x = c$ ，在基本块B2的入口，x的取值有两种可能性： $x \rightarrow \{a, c\}$ 。待分析的基本块B2中的语句 $s: x = b$ ，因对变量x进行了重新赋值，之前对x的赋值全部失效了，于是，我们认为，对变量x的赋值语句 $x = b$ ，“生成”了赋值情况 $x \rightarrow b$ ，“杀死”了其他所有对x的赋值情况。

那么传递函数中 gen_s , $\text{In}[s]$, kill_s 分别为：

gen_s : 语句 $x = b$ 中，对于被赋值的变量x，生成了赋值情况 $x \rightarrow b$ 。

$\text{In}[s]$: 执行语句之前所有变量可能的赋值情况。

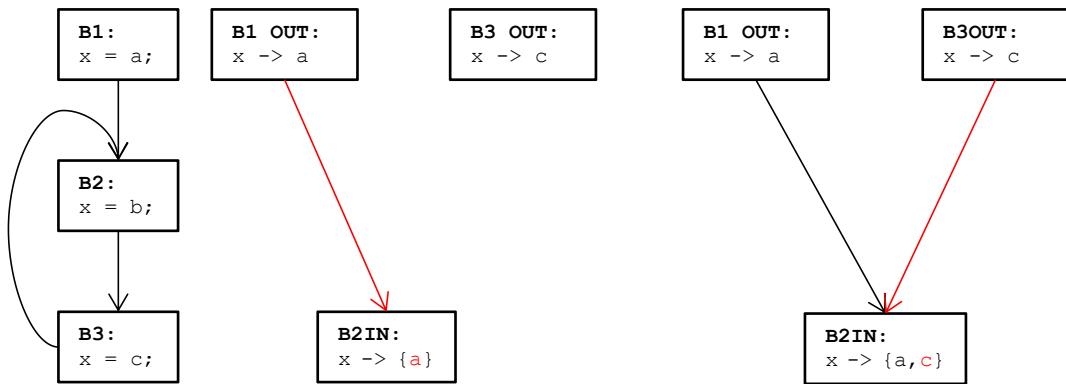
kill_s : 执行语句之后，对于被赋值的变量x，杀死了x当前所有的赋值情况。

则传递函数 $f_s = \text{gen}_s \cup (\text{In}[s] - \text{kill}_s)$ 的语义为，对于待分析的赋值语句s，执行语句之前变量赋值关系记为 $\text{In}[s]$ ，若其对变量x进行赋值，则先使得x的所有赋值关系失效 $(\text{In}[s] - \text{kill}_s)$ ，然后生成当前赋值语句对应的赋值关系 gen_s ，最后将 gen_s 加入到程序状态中。

对于控制流约束函数而言，到达定值模式描述的是在某一程序点上的变量x是否可能恒为一定值，问题可被转化为：x的所有可能存在的赋值情况是否相同。为此，我们通常将控制流约束函数构造为如下的形式：

$$IN[B] = \cup_{P \text{是B的一个前驱}} OUT[P]$$

我们继续将该控制流约束函数运用到上面的例子中，描述程序状态变化。



基本块出口变量x的赋值情况集合的语义为：对于集合内的任一值d，存在至少一条路径，该路径在基本块的出口位置，x的赋值情况为x->d。

B2的前驱基本块共有两个：基本块B1与B3，那么，要分析在基本块B2入口处，x可能的赋值情况，需要合并两个前驱基本块B1和B3在出口处的程序状态。程序内部路径的跳转，可能从B1跳转到B2，也可能从B3跳转到B2，因此对于B2入口处x的赋值集合，应当为B2所有前驱基本块出口位置x赋值的并集。

构造了传递函数与控制流约束函数，接下来我们应当基于函数构造算法，计算我们想要的分析结果。我们用一个简单的程序举例，对其进行常量传播分析。

迭代算法与不动点：

迭代算法是迭代式地进行程序状态计算，直到程序状态到达一不动点的算法。先前我们进行了MFP、MOP、IDEAL solution方法的比较，我们知道，迭代算法的结果较之MOP与IDEAL solution，其获得的程序状态更为保守。

迭代算法的思路是：根据一定的顺序，对程序内部的基本块进行遍历，基于传递函数与控制流约束函数，进行程序状态的改变与记录，当程序状态到达不动点时，迭代算法结束，给出程序内部每个程序点上的程序状态情况。

课本上描述到达定值的迭代算法如下：

```

OUT[ENTRY] = ∅;

for(除ENTRY之外的每个基本块B) OUT[B] = ∅;

while(某个OUT值发生了改变){

    for(除ENTRY之外的每个基本块B){

        IN[B] =  $\cup_{P \text{是} B \text{的一个前驱}} OUT[P]$ ;

        OUT[B] = genB  $\cup$  (In[B] - killB);

    }

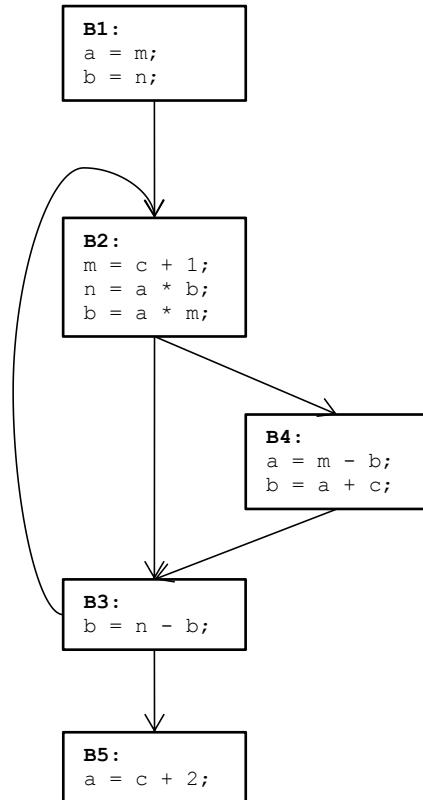
}

```

其中，ENTRY表示程序的入口基本块，算法基于传递函数与控制流约束函数，对程序进行遍历，计算程序状态，直到程序内部的每个基本块在出口时的程序状态均不再改变时停止。

在运用迭代算法计算到达定值之前，我们首先需要关注我们算法的几个性质。你可能心中有这样的疑惑：我们构造的迭代算法一定能到达一个不动点吗？这关乎我们的算法能不能停止，并且得到我们想要的结果。

为了解释这一问题，我们使用计算到达定值的迭代算法作示例。



程序状态的有界性：

上面这段代码，总共有9条赋值语句，与赋值语句相对应的赋值情况也有9条，每一条赋值情况有两种可能状态：alive和dead，如果用一个一维数组来表示每个赋值情况的状态，用0代表dead，用1代表alive，那么(0,0,0,0,0,0,0,0,0)表示9条赋值情况均失效，(1,1,1,1,1,1,1,1,1)表示9条赋值情况均生效，每一个程序点上的状态，至多有 2^9 种情况，程序的状态数量是有界的。

程序状态的单调性：

对于到达定值问题，一程序点p上对应的程序状态中的一条赋值情况 $x \rightarrow a$ ，其语义为：存在至少一条路径，使得到达该程序点p时，x的值为a，使用上文所述的一维数组来表示，即在程序点p上， $x \rightarrow a$ 的状态为1。那么，在迭代遍历的过程中，若当次分析完毕后，某一程序点p上该赋值情况的状态为1，在之后的每一次迭代中，该赋值情况恒为1（存在路径使得赋值情况生效），因此，对于每一条赋值情况，状态的改变仅可能由0变为1，状态的变化是单调的。

因单调而有界，我们的迭代算法最终会到达一不动点，给出分析结果。

遍历顺序：

使用迭代算法进行分析，我们还需了解遍历程序的顺序。在实际的程序执行过程中，程序内部的语句执行是有顺序的，对与程序状态的改变也有先后。对于没有循环的代码，我们或许能轻易地理清基本块之间执行的先后性，然而，对于复杂的带有函数调用和循环的程序，我们应当以一种什么顺序进行分析呢？

在离散数学的学习中，你或许了解过一种排序方式：拓扑排序。拓扑排序用于描述节点之间的先后顺序，我们基于拓扑排序进行程序状态的分析。

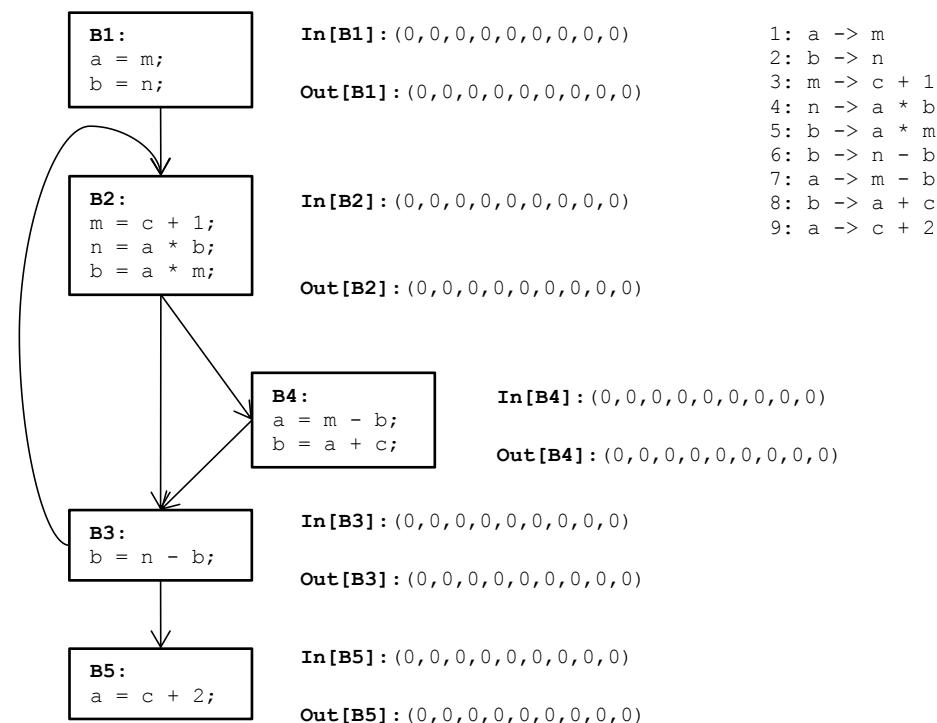
程序控制流图是一个有向图，要构造程序控制流图的拓扑排序，我们可以基于程序控制流图，构造深度优先生成树。课本上介绍了深度优先生成树的构造方法，基于深度优先生成树，我们可以构造节点的有向无环图，从而构造拓扑排序。

有了基本块的拓扑排序，我们即可按照拓扑排序，依序遍历每一个基本块，进行程序状态的分析与计算。

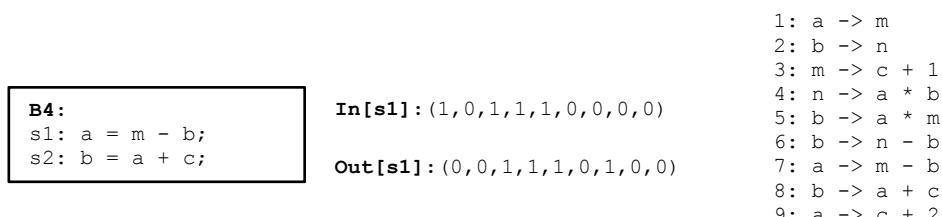
迭代算法计算到达定值：

了解了迭代算法的相关性质，我们接下来用一个例子说明迭代算法进行程序状态计算的整个过程。

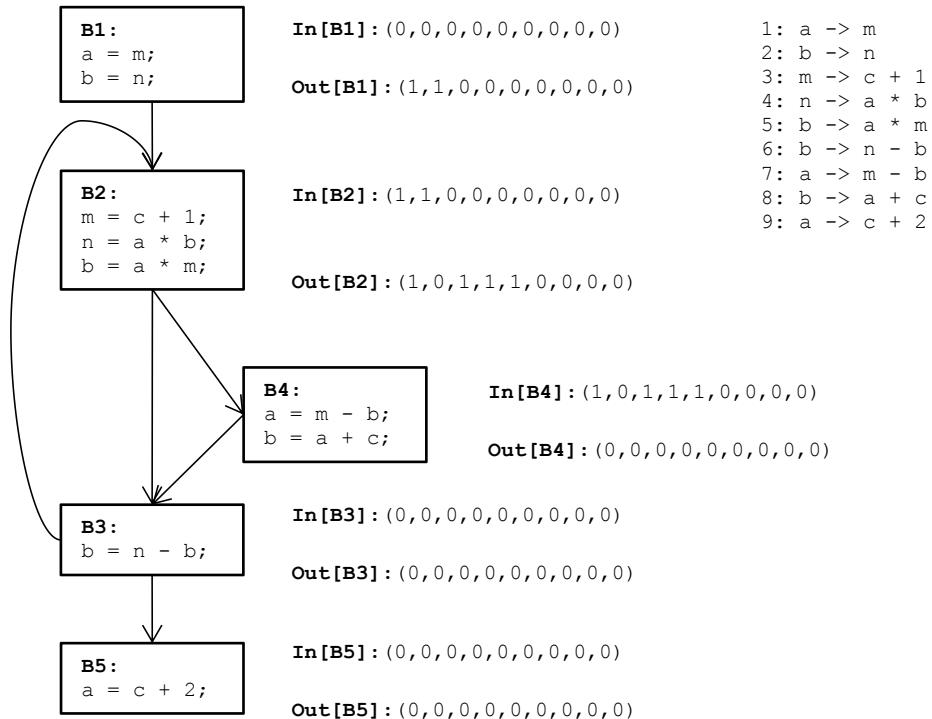
我们使用上文所述的一维数组来表示程序状态的情况。因我们进行到达定值计算，每个程序点上的程序状态的语义为：对于状态为1的赋值情况，存在至少一条经过该程序点的路径，使得该赋值情况生效。因此，在进行分析之前，将程序内的赋值情况均初始化为0。



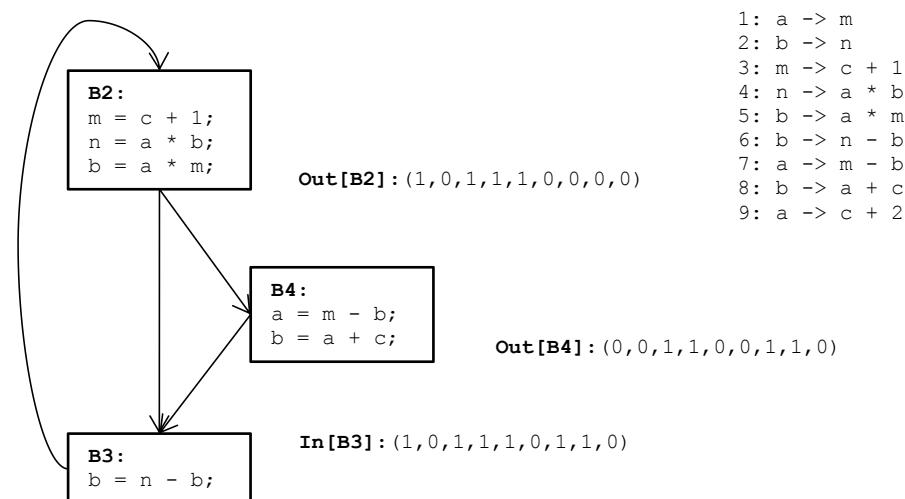
基于拓扑排序生成算法与深度优先遍历，程序的遍历顺序为：B1, B2, B4, B3, B5。在遍历的过程中，使用传递函数与控制流约束函数进行程序状态的计算，以下是运用传递函数与控制流约束函数的示例：



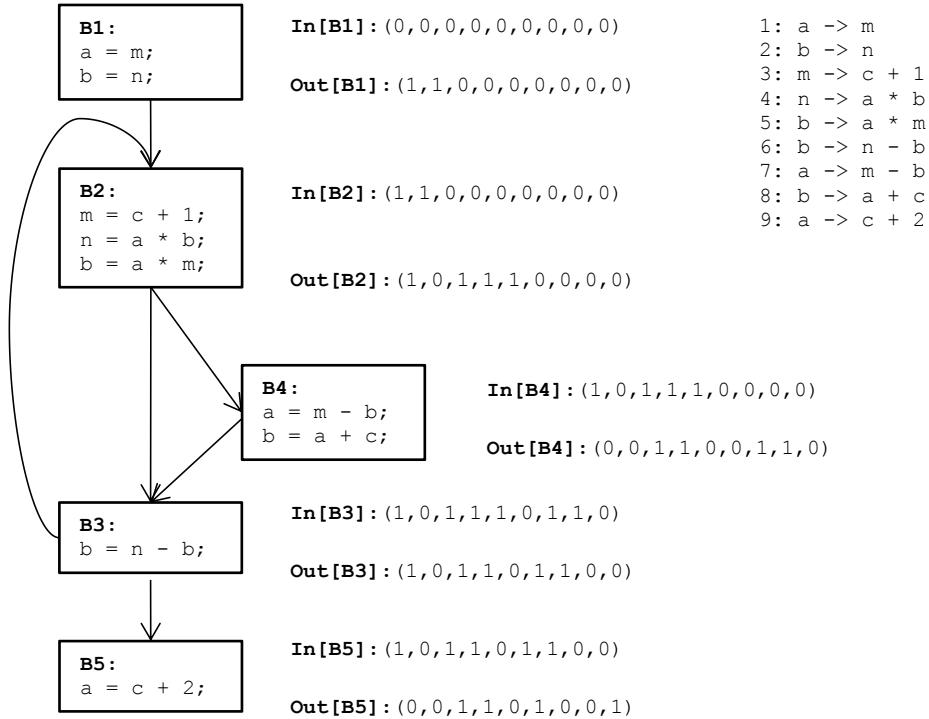
In[s1]与OUT[s1]表示在执行语句s1之前和之后的程序状态。在执行s1之前，赋值情况1、3、4、5生效，s1语句对变量a赋值，首先杀死所有变量a的赋值情况（1、7、9），然后生成s1对应的赋值情况7，因此，在执行s1后，赋值情况3、4、5、7生效。



对于基本块B3，其有两个前驱基本块B2和B4，由B2向B3传递的程序状态为(1,0,1,1,1,0,0,0,0)，由B4向B3传递的程序状态为(0,0,1,1,0,0,1,1,0)，使用OR运算合并程序状态，得出在B3入口处，程序状态为(1,0,1,1,1,0,1,1,0)。

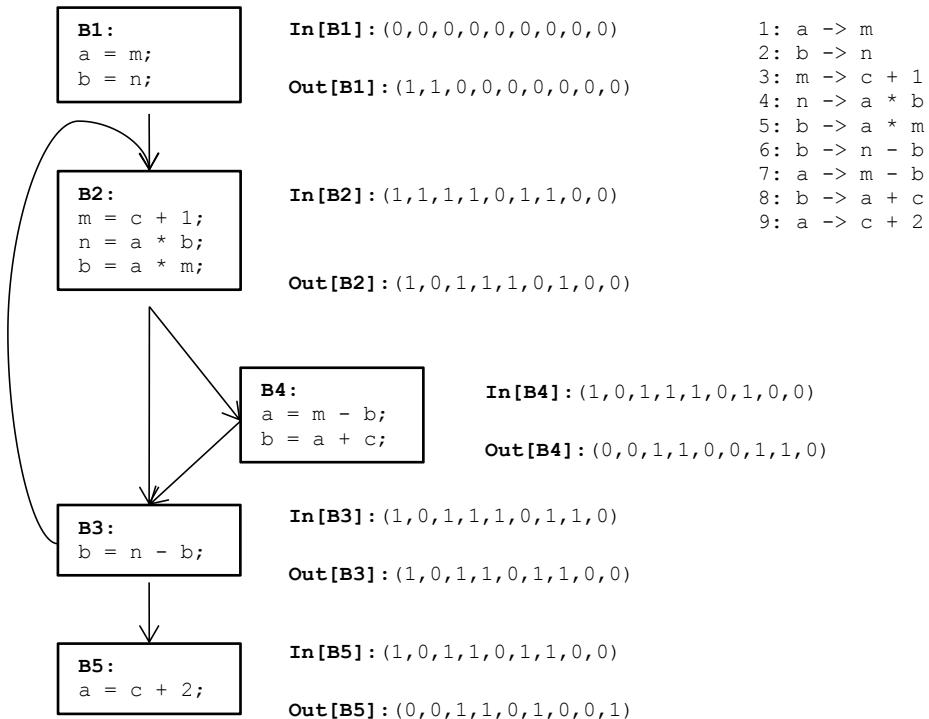


那么，第一次遍历后记录的程序状态如下所示：



根据迭代算法，每一次迭代后若有基本块的OUT发生了变化，则进行下一次遍历。

第二次遍历如下图所示：



第二次遍历后，仍有基本块的OUT发生了变化，因此进行第三次遍历。

在第三次遍历后，相较于第二次遍历的结果，没有任何基本块的OUT值发生了改变。

因此，迭代算法结束，输出分析结果。从记录的程序状态中，能够获取我们想要的程序性质，例如，对于基本块B₃中的语句 $a = m - b$ ，赋值语句右侧的表达式由变量m与b构成，对于变量m，生效的赋值关系为 $m \rightarrow c + 1$ ，对于变量b，生效的赋值关系为 $b \rightarrow a * m$ ，若均为常量，则可将变量a也用一常量替换。

在遍历的过程中，你可能会发现，部分对于程序状态的计算是不必要的，并且你可能也会有疑惑：为什么算法的停止条件是没有任何一个基本块的OUT产生改变。

显然，如果一个基本块的前驱基本块只有一个，那么直接拷贝前驱基本块的OUT状态，作为基本块的IN即可，那么若前驱节点的OUT不变，基本块的IN也不变。若存在多个前驱，且OUT均不变，在进行或运算时，得出的结果也不变。

当一个基本块的IN状态不变时，因基本块内部的语句数量有限且不变，那么对于程序状态的改变也不变，因此当IN状态不变时，其OUT也不变。

于是，当一个基本块的IN状态不变时，没有必要重新计算程序状态的改变，可以简单地拷贝上一轮分析的OUT，作为本次分析的结果。

为了去除算法内部的部分冗余计算，我们将迭代算法修改为工作表算法(worklist algorithm)，使用一个worklist记录需要进行状态计算的基本块，算法修改如下：

```
OUT[ENTRY] = ∅;  
for(除ENTRY之外的每个基本块B) OUT[B] = ∅;  
Worklist <- 所有的基本块;  
while(Worklist非空){  
    从工作表中选择一个基本块B  
    OLD_OUT = OUT[B];  
    IN[B] =  $\cup_{P \text{是} B \text{的一个前驱}} OUT[P]$ ;  
    OUT[B] = genB  $\cup (In[B] - kill_B)$ ;  
    if (OLD_OUT ≠ OUT[B])  
        把基本块B的所有后继加入到工作表中}
```

}

相较于迭代算法，工作表算法多了添加基本块进入工作表的语句，当一个基本块的输出发生改变时，需要将其后继加入到工作表中，进行程序状态的更新与计算，算法的停止条件是工作表为空。

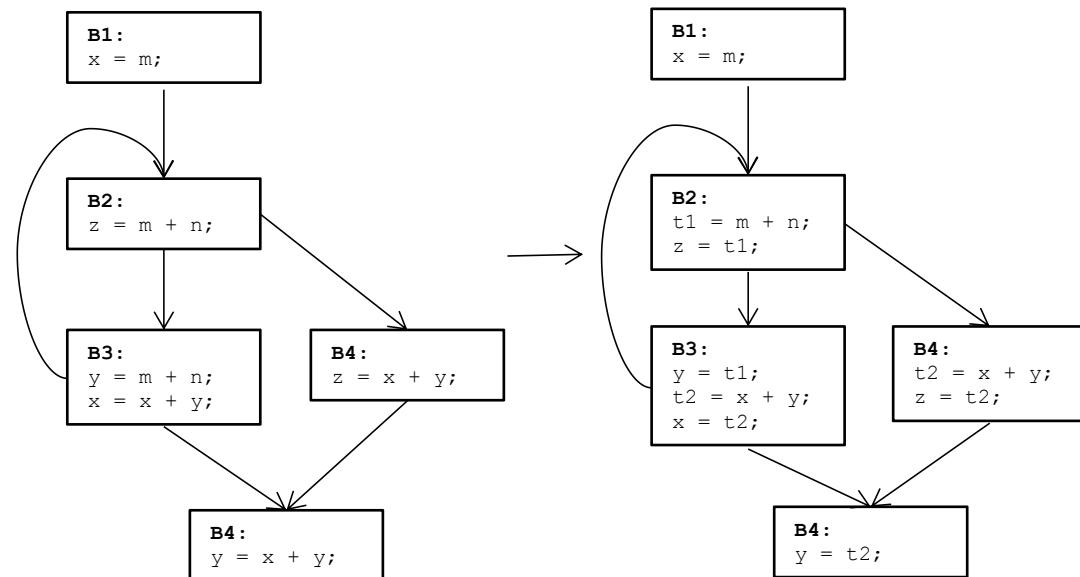
相比迭代算法遍历程序的每个基本块，工作表算法第一次遍历计算了基本块B1、B2、B3、B4、B5，第二次计算了基本块B2、B3、B4、B5，第三次仅需计算B3与B4之后，算法终止，输出结果。

在实验五中，你需要合理地使用到达定值模式，进行部分优化任务的分析与计算。

6.2.6 数据流分析模式（可用表达式）

可用表达式模式(available expression)关注程序内部的表达式是否在某些程序点可用。表达式可能由多个变量构成，表达式的计算过程包含了变量的use，表达式处于赋值语句右侧，包含了变量的define，与程序内部的变量的define-use关系相关，较上文提到的到达定值问题更为复杂一些。

在程序优化的过程中，可用表达式模式通常用于进行**公共子表达式消除(common subexpression elimination)**。通过可用表达式模式对函数内部进行分析，消除重复计算的公共子表达式：

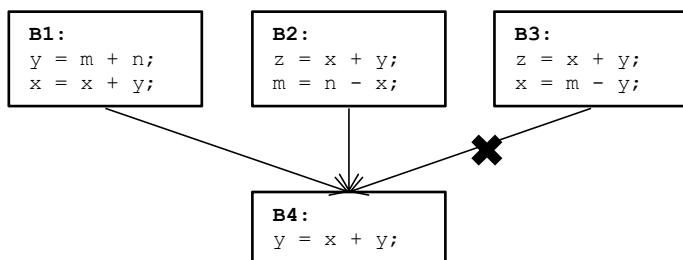


表达式 $m+n$ 与 $x+y$ ，在程序内部被多次计算，因此可以使用 t_1 与 t_2 代替其求值的结果，简化表达式的计算。

可用表达式模式应用：

与到达定值模式相同，首先我们确定应当使用may分析还是must分析进行程序状态的分析与计算。

可用表达式问题，用于分析在某一程序点p上的表达式e，对于所有经过去程点p的路径，表达式是否均已被计算过，且构成表达式的变量在之后未被重新定义，即，我们需判断所有经过p的路径，在程序点p上该表达式均存活。为此，我们使用must分析的方式进行程序状态的分析。



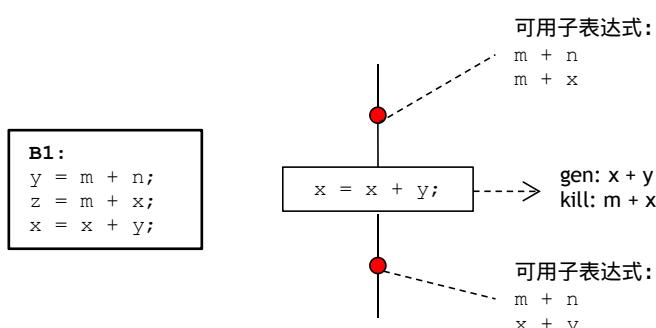
例如，对于上面这段代码，在基本块B3中对 $x+y$ 进行了计算，但是之后对变量x的赋值使得表达式的计算结果失效，因此认为B3在出口处该表达式不存活。

然后，我们要构造针对可用表达式问题的传递函数与控制流约束函数。

对于可用表达式问题，我们需要关注其中的表达式与构成表达式的变量的赋值语句。对于表达式 $b + c$ ，构成表达式的变量有b和c两个，对于语句s1: $a = b + c$ ，其中包含了对 $b + c$ 的求值，使得 $b + c$ 生效，若存在对b的赋值语句，s2: $b = d$ ，对于变量b进行重新定义，使得表达式 $b + c$ 的求值失效。于是，对于表达式 $b + c$ ，其状态转换的传递函数定义为：

$$f_s = e_gen_s \cup (In[s] - e_kill_s)$$

我们将该传递函数运用到如下的程序中，描述程序状态的变化：



在执行语句 $x = x + y$ 之前，程序内部的可用子表达式共有两个： $m + n$ 与 $m + x$ ，语句 $x = n + y$ 生成了子表达式 $n + y$ ，同时对 x 重新进行赋值。因表达式 $m + x$ 中存在对 x 的使用，对 x 的重新赋值使得 $m + x$ 的求值失效。于是，我们认为，语句 $x = n + y$ ，“生成”了可用子表达式 $n + y$ ，“杀死”了可用子表达式 $m + x$ 。

那么传递函数中 e_{gen}_s , $\text{In}[s]$, e_{kill}_s 分别为：

e_{gen}_s : 语句 $x = n + y$ 中，生成了可用子表达式 $n + y$ 。

$\text{In}[s]$: 执行语句之前所有可用子表达式。

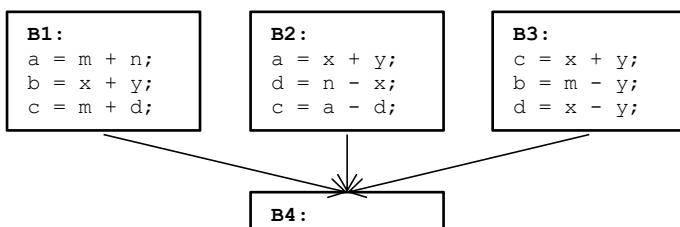
e_{kill}_s : 执行语句之后，杀死的可用子表达式。

则传递函数 $f_s = e_{\text{gen}}_s \cup (\text{In}[s] - e_{\text{kill}}_s)$ 的语义为，对于待分析的赋值语句 s ，执行语句之前存在的子表达式情况记为 $\text{In}[s]$ ，若其对某变量 x 进行赋值，则先使得包含 x 的所有子表达式失效 ($\text{In}[s] - e_{\text{kill}}_s$)，然后生成当前语句对应的子表达式 e_{gen}_s ，最后将 e_{gen}_s 加入到程序状态中。

对于控制流约束函数而言，可用子表达式模式描述的是在某一程序点上是否在任何情况下某一子表达式 e 均生效。因其为must分析，我们通常将控制流约束函数构造为如下的形式：

$$\text{IN}[B] = \cap_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P]$$

我们将该控制流约束函数运用到下面的例子中，描述程序状态变化。



$$\begin{aligned}
 \text{B4_IN} &= \text{B1_OUT} \cap \text{B2_OUT} \cap \text{B3_OUT} \\
 &= \{m + n, x + y, m + d\} \cap \{x + y, n - x, a - d\} \cap \\
 &\quad \{x + y, n - x, a - d\} \\
 &= \{x + y\}
 \end{aligned}$$

B_4 的前驱基本块共有三个：基本块 B_1 、 B_2 与 B_3 ，那么，要分析在基本块 B_4 入口处生效的子表达式，需要合并前驱基本块在出口处的程序状态。要保证子表达式在基本块 B_4 入口仍生效，需确保所有的前驱基本块的出口处该子表达式生效。因此，可用子表达式的控制流约束函数，使用与运算进行程序状态的合并。

构造了传递函数与控制流约束函数，接下来我们应当基于函数构造算法，计算我们想要的分析结果。

```

OUT[ENTRY] = ∅;

for(除ENTRY之外的每个基本块B) OUT[B] = ∅;

while(某个OUT值发生了改变){

    for(除ENTRY之外的每个基本块B){

        IN[B] = ∩ P是B的一个前驱 OUT[P];

        OUT[B] = e_gens ∪ (In[s] - e_kills);

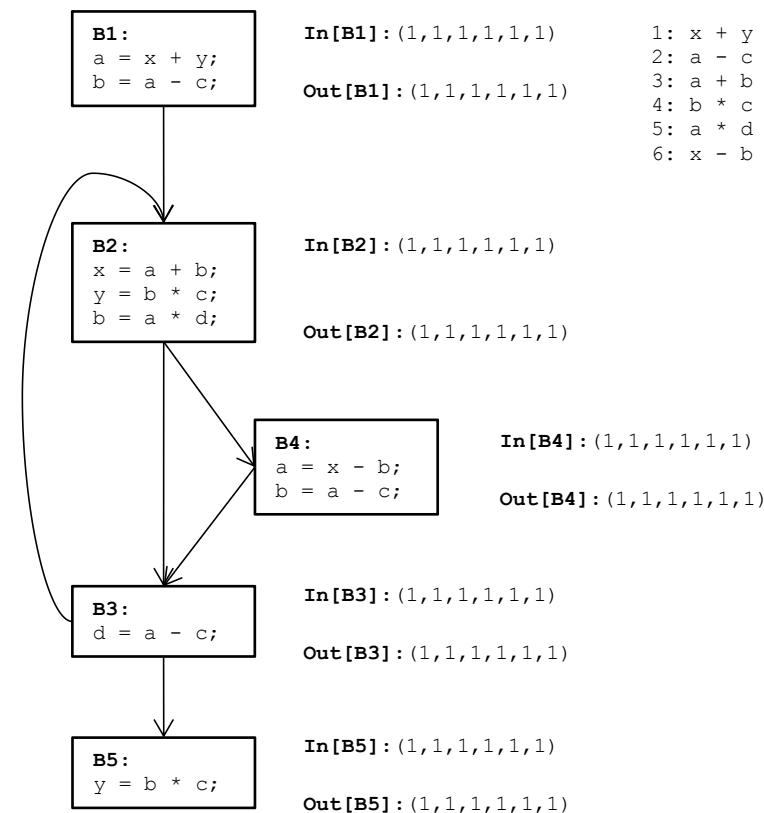
    }
}

```

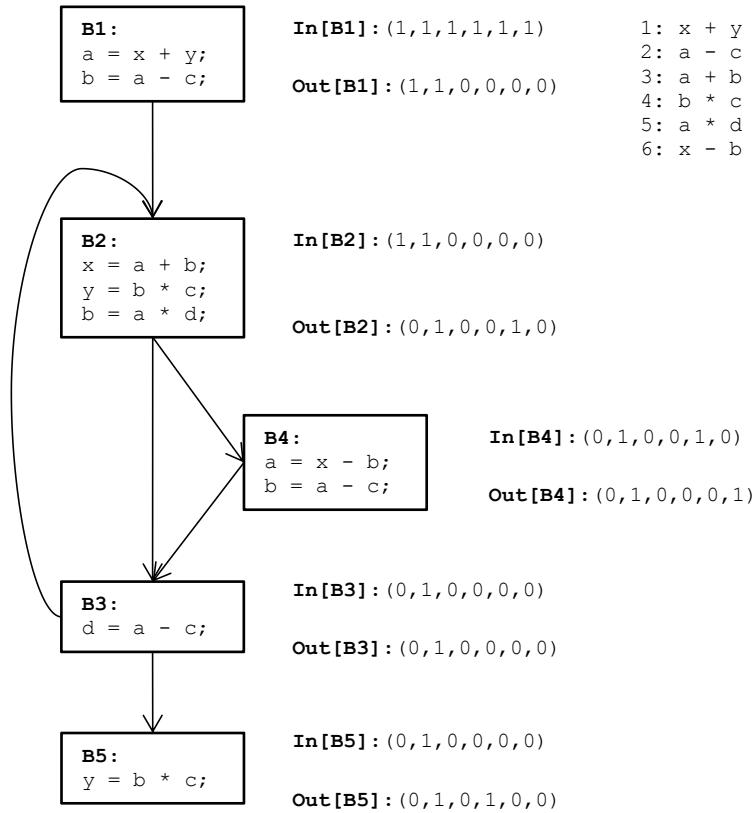
我们用一个简单的程序举例，对其进行可用子表达式分析。

因使用must分析方式，只需有一个前驱节点不包含子表达式，在基本块的入口该表达式即处于失效状态，因此我们在进行分析之前，将程序内的子表达式生效情况均初始化为1。

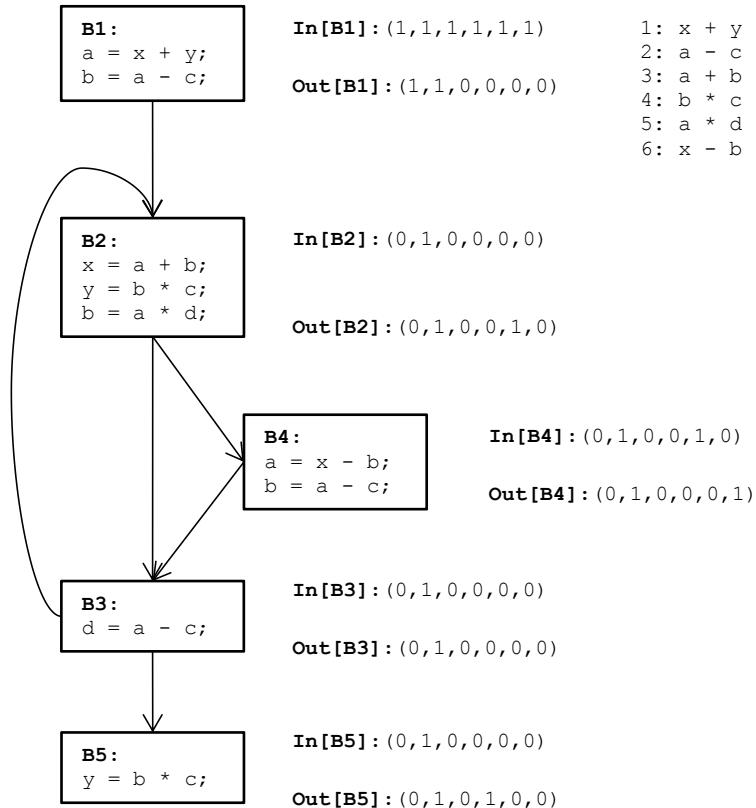
因可用表达式的有效关系与程序执行顺序一致，因此遍历程序的顺序与到达定值模式一致。



那么，第一次遍历后记录的程序状态如下所示：



第二次遍历后记录的程序状态如下所示：



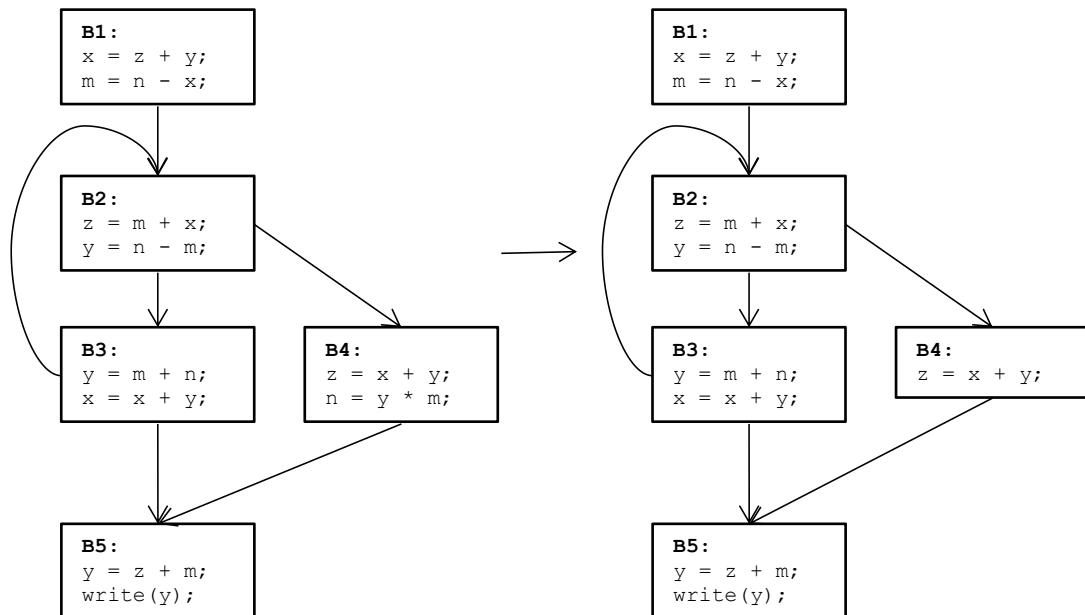
在第二次遍历后，每个基本块的OUT值均没有改变，因此迭代算法就此终止。

在实验五的实现中，你可以选用迭代算法，也可以使用如6.2.5节所示的工作流算法，实现优化算法。

6.2.7 数据流分析模式（活跃变量）

活跃变量分析(live-variable analysis)关注程序内部程序点 p 上对于某变量的定义，是否会在某条由 p 出发的路径上被使用。活跃变量分析与程序内部的变量的define-use关系相关，当变量被define后未被use，该define语句被认为是无用的。

在程序优化的过程中，活跃变量模式通常用于进行**无用代码消除(dead code elimination)**。通过活跃变量模式对函数内部进行分析，消除未被使用的赋值与未执行的程序代码：



基本块B4中的语句 $n = y * m$ ，对于变量 n 的define在之后的程序中未被use，因此这个define语句是无用的，可消除此处对于变量 n 的赋值，优化程序代码。

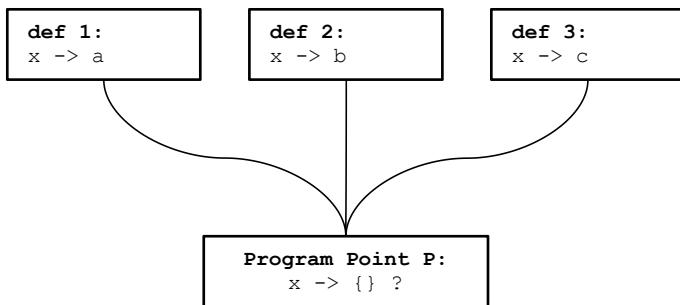
活跃变量模式应用：

首先我们确定应当使用may分析还是must分析进行程序状态的分析与计算。

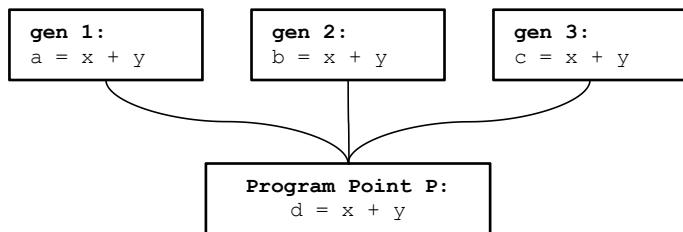
活跃变量问题，用于分析在某一程序点 p_1 上对变量 v 的赋值，是否在之后的程序点 p_2 上被使用，且在 p_1 到 p_2 的路径上没有对该变量 v 的重定义，即，我们需判断所有经过 p_1 的路径，是否存在一条路径使得变量 v 在之后被使用。为此，我们使用may分析的方式进行程序状态的分析。

然后，我们要构造针对活跃变量问题的传递函数与控制流约束函数。

活跃变量的遍历方式与之前的到达定值与可用表达式的遍历有所不同：到达定值问题与可用表达式问题使用一种叫做前序遍历的遍历方式，而活跃变量问题要用一种后序的遍历进行程序状态的分析。

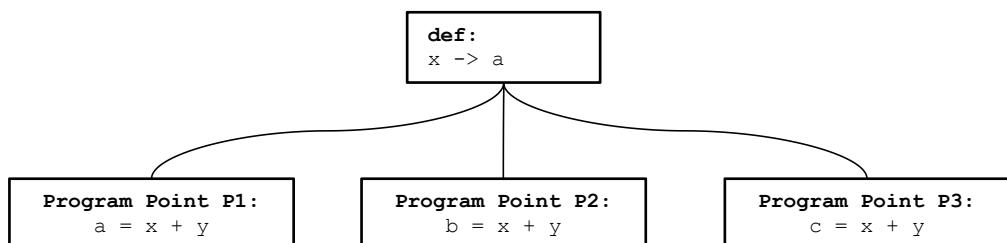


到达定值模式分析某一程序点p上对某一变量x的所有可能赋值情况，对应多条经过p的路径上前驱基本块中对变量x的赋值语句。



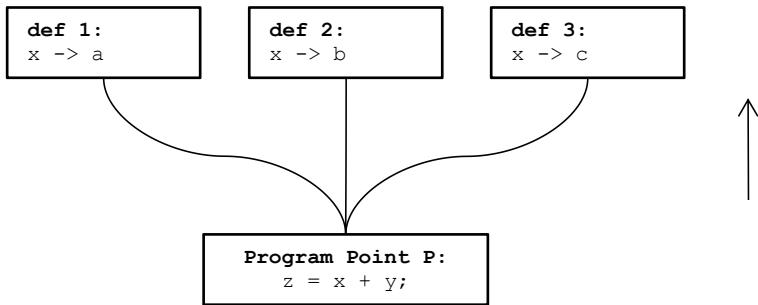
与到达定值模式相同的是，可用子表达式模式分析在某一程序点p上是否存在可用的子表达式，子表达式在多条经过程序点p的前驱基本块内部被计算，算法分析在程序点p处被计算的子表达式是否还生效。

然而，对于活跃变量分析，虽然也是与变量的define-use相关的分析，其分析针对的是某次对变量的define是否生效，如下所示：



对变量x的一次定义 $\text{def: } x \rightarrow a$ ，可能在后继基本块中的多个程序点p1、p2、p3等被使用，而多个定义可能在同一个程序点上被使用，因此使用前序遍历较为复杂。

因此，我们使用后序遍历的方式进行数据流分析，将问题转化为如下的情况：

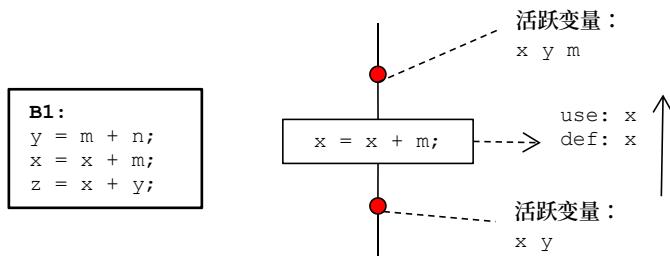


对于程序点P上的语句，对x进行了使用(use)，那么我们分析这一条语句使得哪些前驱基本块内部对于变量x的赋值(define)不是无用代码。我们使用后序遍历的方式，因此我们构造的传递函数与控制流约束函数较之前两种分析模式也有一些不同。

对于活跃变量问题，我们需要关注变量的定义(def)与使用(use)语句。使用后序遍历时，遇到语句 $z = x + y$ ，表示前驱基本块中有对变量x与y的定义语句生效，当继续遍历遇到对变量x的赋值语句时，完成一组def-use关系。于是，对于程序内部的表达式，其状态转换的传递函数定义为：

$$f_s = \text{use}_s \cup (\text{OUT}[s] - \text{def}_s)$$

我们将该传递函数运用到如下的程序中，描述程序状态的变化：



语句 $x = x + m$ 中对x进行了定义，并在语句 $z = x + y$ 中，该定义被使用。因此， $x = x + m$ 首先将变量x从活跃变量列表中移除，然后，对于x的赋值使用到了x和m两个变量，因此这两个变量将在上文被定义，所以将变量x与m加入到活跃变量列表中。于是，我们认为，语句 $x = x + m$ ，从活跃列表中移除了变量x，添加了变量x与m。

那么传递函数中 e_{gen}_s , $\text{In}[s]$, e_{kill}_s 分别为：

use_s : 语句 $x = x + m$ 中，使用了变量x与m。

$\text{OUT}[s]$: 执行语句后的程序状态。

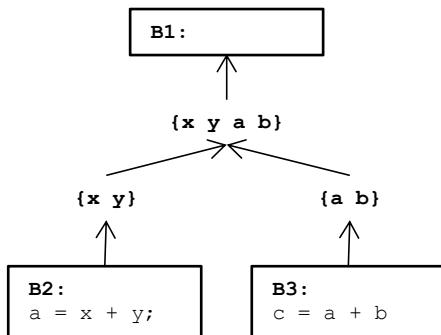
def_s : 语句 $x = x + m$ 中，定义的变量x。

则传递函数 $f_s = \text{use}_s \cup (\text{OUT}[s] - \text{def}_s)$ 的语义为，对于待分析的赋值语句 s ，执行语句之后存在的赋值情况记为 $\text{Out}[s]$ ，若其对某变量 x 进行赋值，则先从赋值语句中去除该变量 x ($\text{OUT}[s] - \text{def}_s$)，然后将当前语句使用的变量 use_s 加入到程序状态中。

对于控制流约束函数而言，活跃变量模式描述的是在某一程序点上，其后继的基本块中是否可能存在对某变量的使用。因其为may分析，我们通常将控制流约束函数构造为如下的形式：

$$\text{OUT}[B] = \bigcup_{S \text{ 是 } B \text{ 的一个后继}} \text{IN}[S]$$

我们将该控制流约束函数运用到下面的例子中，描述程序状态变化。



B_1 的后继基本块有两个：基本块 B_2 和 B_3 ，要分析在基本块 B_1 出口处活跃的变量，需要合并后继基本块在入口处的程序状态。基本块 B_2 和 B_3 分别提供了活跃变量 x, y 和 a, b ，因变量在后继基本块 B_2 与 B_3 中被使用，所以变量均可能在 B_1 中存在定义语句。因此，活跃变量模式的控制流约束函数，使用或运算进行程序状态的合并。

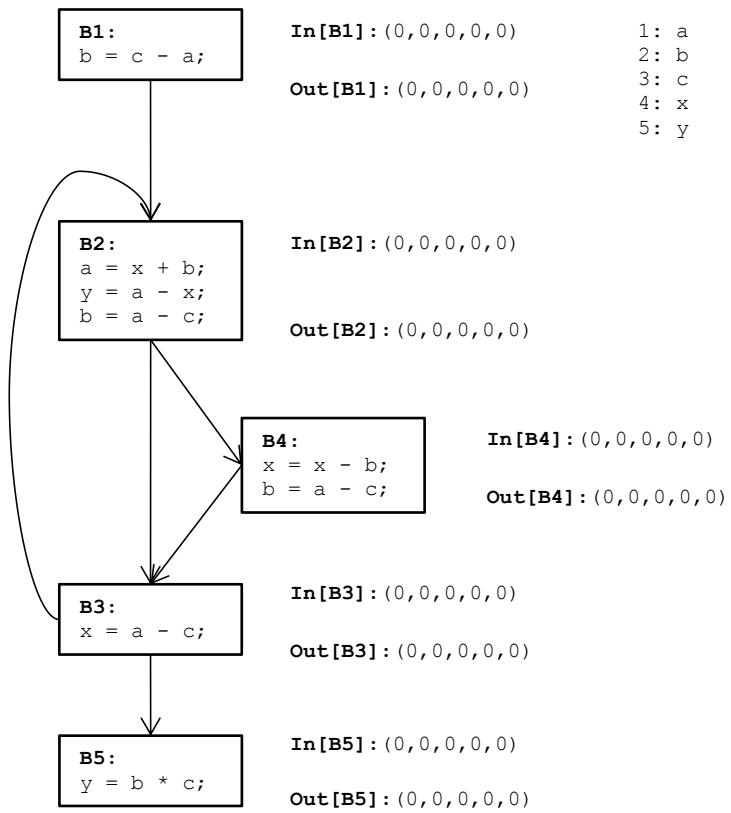
构造了传递函数与控制流约束函数，接下来我们应当基于函数构造算法，计算我们想要的分析结果。我们用一个简单的程序举例，对其进行活跃变量分析。

因活跃变量分析使用后序分析的方式，因此构造的迭代算法与之前的有所不同：

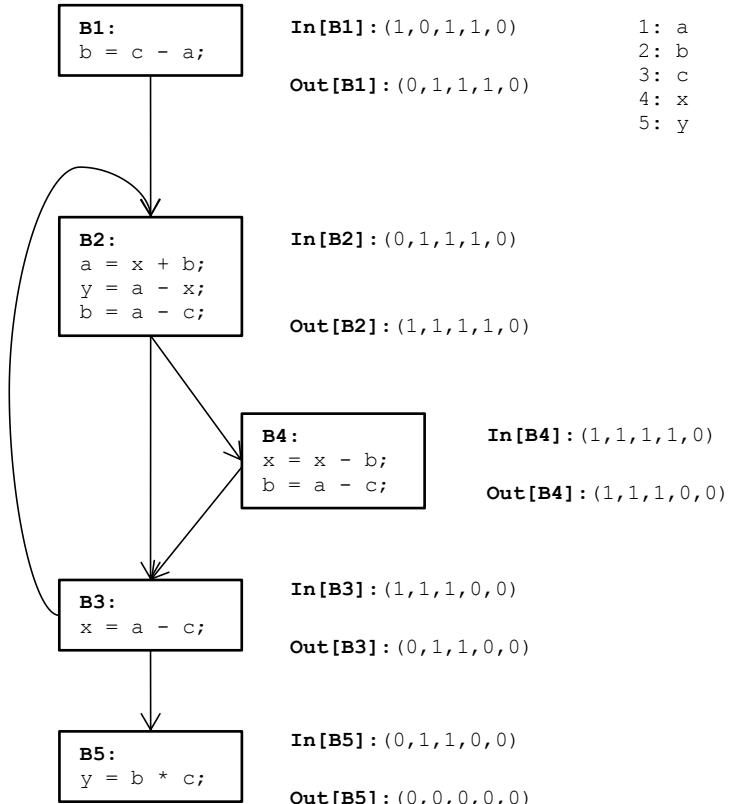
```

IN[EXIT] = ∅;
for(除EXIT之外的每个基本块B) IN[B] = ∅;
while(某个IN值发生了改变){
    for(除EXIT之外的每个基本块B){
        OUT[B] = ∪ S是B的一个后继 IN[S];
        IN[B] = use_B ∪ (OUT[B] - def_B);
    }
}
  
```

因使用may分析方式，我们在进行分析之前，将程序内的变量活跃情况初始化为 o 。



那么，第一次遍历后记录的程序状态如下所示：



在第二次遍历后，每个基本块的IN值均没有改变，因此迭代算法就此终止。

在实验五的实现中，你需要使用上述的三个数据流分析模式，实现不同的优化策略，你可以选择实现其中的一个或多个，完成优化管道的构建。

6.2.8 全局优化

前文提到，良好的优化顺序能够降低编译过程中的时空开销，针对生成的中间代码，进行了局部优化之后，我们构造全局优化管道：**常量传播-公共子表达式消除-常量折叠-控制流优化-无用代码消除-循环不变代码外提-归纳变量强度削减-控制流优化**，我们使用先前所示的程序与控制流图进行全局优化。

全局优化1 常量传播：

我们之前在进行局部优化时，将恒为常量的变量转化为常量。基本块内部语句都按照从入口到出口的顺序依次执行，因此进行常量折叠并不困难。那么，对于在多个基本块之间乃至复杂的控制流图中进行常量传播的计算应当采取什么样的方法呢？

常量传播框架与到达定值模式较为接近，所不同的是，到达定值中对于变量的赋值情况只存在两种状态：生效与失效，而对于常量传播框架而言，常量值的集合是无界的。

常量传播框架中的变量的状态分为三种：

1. 所有符合该变量类型的常量值
2. NAC, not-a-constant，表示当前变量不是一个常量值。这代表该变量在到达程序点p的不同的路径上的值不同，或是被赋予了一个输入变量的值。
3. UNDEF，表示未定义的值。在到达程序点p的不同路径上存在至少一条路径未对变量的值进行定义。

常量传播函数：

三种状态分别对应三种情况：假设程序点上有变量v，第一种状态表示该变量v唯一地对应一个定值a，即在程序点p上v的可能取值集合为{a}，第二种状态NAC表示，在经由程序点p的多条路径上分别对变量v进行赋值，且不同路径到达p时v的可能取值至少有1个且数量有限，即v的可能取值集合为{a,b,c,...,n}，第三种状态UNDEF表示在程序点p上存在至少一条从入口到程序点p的路径，在这条路径上，没有对变量v的赋值或已被赋空。

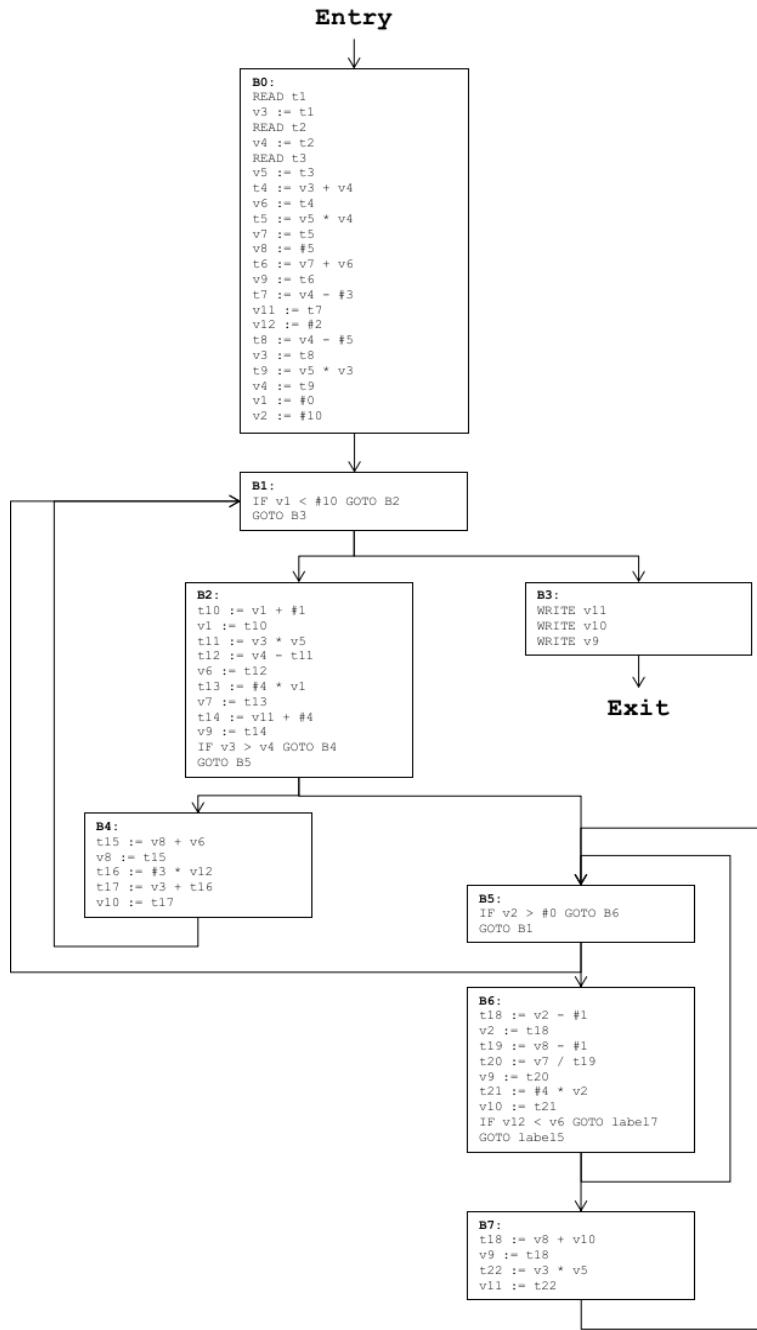


图13 控制流图

常量的传播主要通过变量的赋值语句进行，如 $z = x + y$ 等表达式。设 S_z , S_x , S_y 分别为执行该语句后变量 z , x , y 的状态，则 S_z 的求值存在9种情况：

	UNDEF	c1	NAC
UNDEF			
c2			
NAC			

因其存在对称性，故我们只要讨论其中4种情况即可。

1. x与y中有至少一个变量状态为UNDEF，这表示x与y中包含一个输入变量，且存在一条从入口到程序点p的路径，使得该变量未被重新定义，因其为输入值，所以可以是该类型的任意值，故， S_z 为UNDEF。

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF		
NAC	UNDEF		

2. x为一定值c1, y为一定值c2, 那么 $z = x + y$ 可转化为 $z = c1 + c2$, S_z 为 $c3=c1+c2$ 。

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF	c3	
NAC	UNDEF		

3. x为一定值c1, y为NAC, 即y的取值集合中存在多个值, $y_1 y_2 \dots$, 则 $x+y$ 可转化为 $c1+y_1, c1+y_2 \dots$, z的取值集合也有多个值, 因此 S_z 的状态为NAC。

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF	c3	NAC
NAC	UNDEF	NAC	

4. x, y的状态均为NAC, 则由3可知其状态为NAC。

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF	c3	NAC
NAC	UNDEF	NAC	NAC

根据上述的四种情况可以看出，三种状态的转变具有单调性，状态转变的顺序只可能由UNDEF->常量->NAC，或直接由UNDEF转化为NAC，而不可能反向转化。通过常量传播框架和函数，可构造相应的传递与控制流约束函数，进行程序状态的分析。

我们关注四种常量传播算法：

1) 简单常量传播(simple constant propagation)

课本上所示的常量传播框架由Kildall设计，基于数据流和传递函数进行常量传播优化。算法的设计与前文所介绍的数据流分析模式有些类似：对程序状态进行初始化，构造传递函数与控制流约束函数，使用迭代算法或工作列表算法进行数据流值的传递与计算，输出结果。

传递函数：常量传播框架的传递函数较为复杂，因其存在三种状态与九种情况。根据上文的分析，我们现在明白能够将九种情况归纳为四种，即，对于一条针对变量v的赋值语句s：

1. s的RHS为一常量c，那么x的取值集合为{c}，状态为常量。
2. s的RHS为read函数，那么x的状态为NAC。
3. s的RHS为x+y，则根据上述的四种情况进行计算。
4. s的RHS包含函数调用或其他语言特性，根据你的框架的构造规则进行状态变更计算。

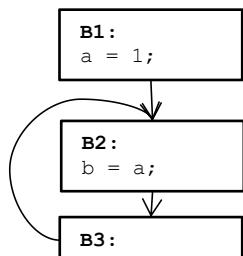
控制流约束函数：

控制流约束函数与传递函数略有不同：

$$\text{UNDEF} \cup c = c \quad \text{NAC} \cup c = \text{NAC}$$

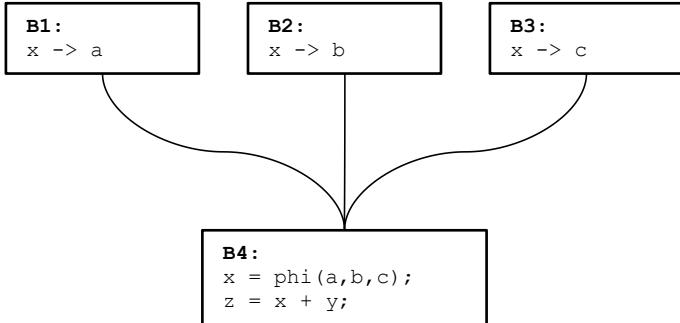
$$c \cup c = c \quad c_1 \cup c_2 = \text{NAC}$$

实验五输入的中间代码，其中不包含对未定义的变量的使用。程序内部的变量状态初始化为UNDEF，因此控制流约束函数与传递函数的一个不同即在于， $\text{UNDEF} \cup c = c$ ，其原因如下图：



我们将程序内部的状态初始化为UNDEF，那么，若 $\text{UNDEF} \cup c = \text{UNDEF}$ ，B3中无对变量a的赋值语句，则在B2的入口，a的状态恒为UNDEF。

我们引入phi函数来表示合并不同前驱基本块上程序状态的函数：



我们在基本块B4中添加phi语句来表现该函数的功能：合并前驱基本块中变量x的程序状态。我们将基本块B1、B2、B3中的变量x依次编号为x1、x2、x3，那么我们运用phi函数进行合并：

x1的取值集合为{a}，x2的取值集合为{b}，x3的取值集合为{c}，x1, x2, x3的状态均为常量。

那么，phi函数依次合并不同路径上传递而来的程序状态，其步骤如下：

0. x_4 的取值集合初始化为 $\{\}$ ，状态初始化为UNDEF。

1. 合并B1->B4，将 x_1 的取值集合加入到 x_4 中， x_4 的取值集合 $\{a\}$

合并 x_1 与 x_4 的状态：常量 (x_1) \cup UNDEF (x_4) = 常量。

2. 合并B2->B4，将 x_2 的取值集合加入到 x_4 中， x_4 的取值集合 $\{a, b\}$

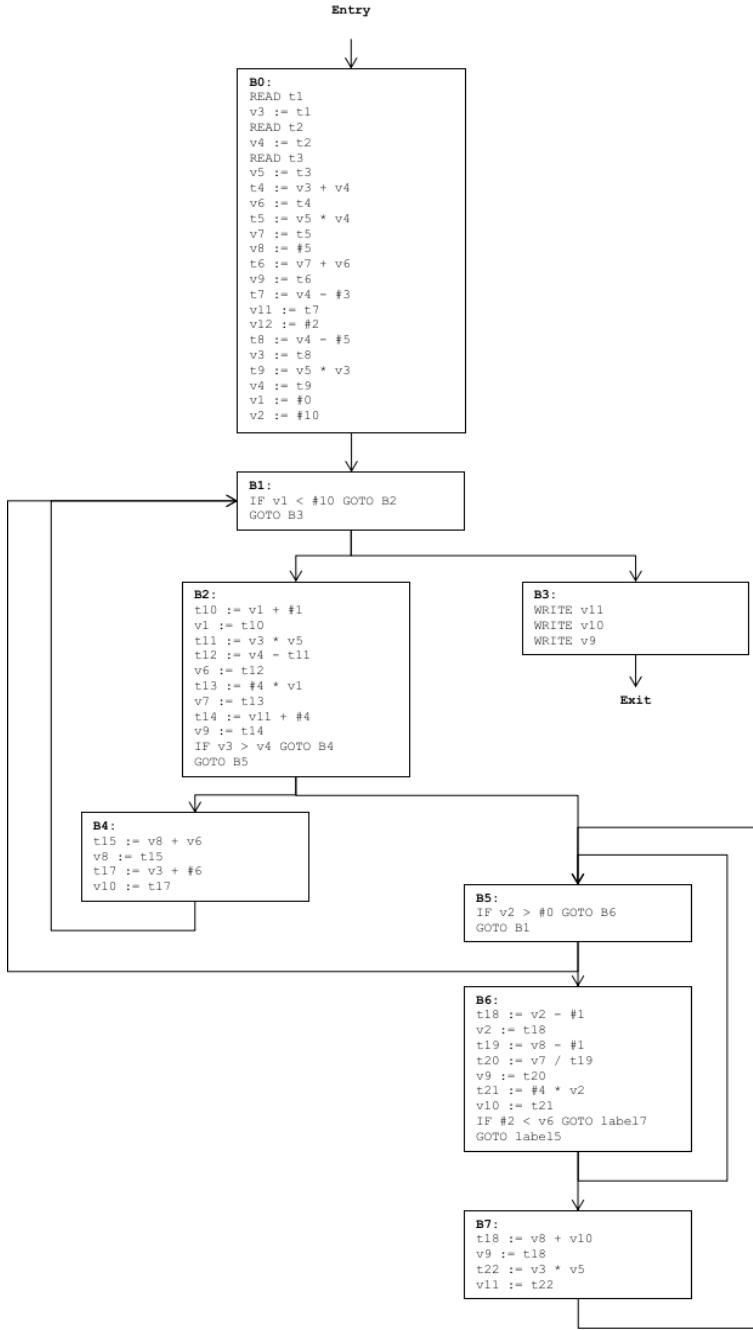
合并 x_2 与 x_4 的状态：常量 (x_2) \cup 常量 (x_4) = NAC。

3. 合并B3->B4，将 x_3 的取值集合加入到 x_4 中， x_4 的取值集合 $\{a, b, c\}$

合并 x_3 与 x_4 的状态：常量 (x_3) \cup NAC (x_4) = NAC。

使用的迭代算法已在数据流分析模式中反复提及，此处不再赘述。

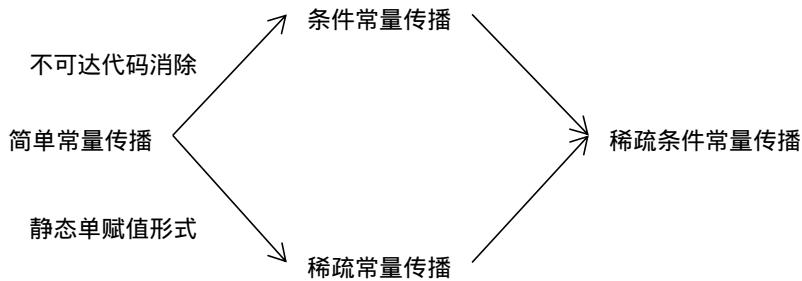
简单常量传播是课本上使用的常量传播框架，因此，在实现实验五中的常量传播算法时，你可使用书上介绍的常量传播框架，基于数据流、传递函数、控制流约束函数与迭代算法，进行常量传播的计算，其计算结果如下：



使用以上的常量传播框架进行计算之后，你的算法应当能够发现，变量 v_{12} 在循环内部的值为一常量，因此可以使用常量去替换。如果你设计的框架足够“聪明”，或许能够发现，基本块 B2 中，定值 t_{12} 的值为零，且执行语句 $v_6 := t_{12}$ 后变量 v_6 的值也恒为零，可进行大量常量替换，简化控制流图等，这需要我们进行一系列的优化，在后文进行叙述。

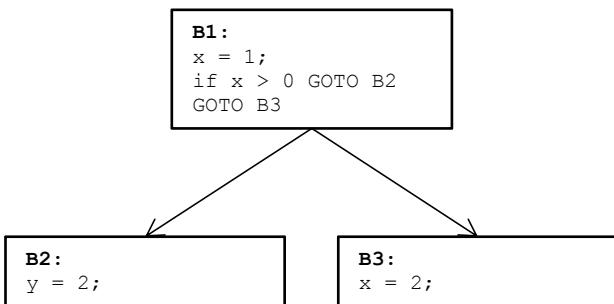
实验五只要求你基于简单常量传播的方式，使用数据流分析，设计传递函数与控制流约束函数构造常量传播优化模块，完成常量传播优化。

在简单常量传播的基础上，我们介绍另外三种常量传播算法。



2) 条件常量传播(conditional constant propagation)

你可能会发现，在某些情况下，数据流图上的部分基本块实际上是不可达的。这些不可达的基本块中，有一部分是可以在编译阶段进行代码削减，这部分代码属于无用代码消除中的一部分，被称为**不可达代码消除 (unreachable code elimination)**。不可达基本块中对变量的赋值关系，不应当加入到常量传播的计算之中。

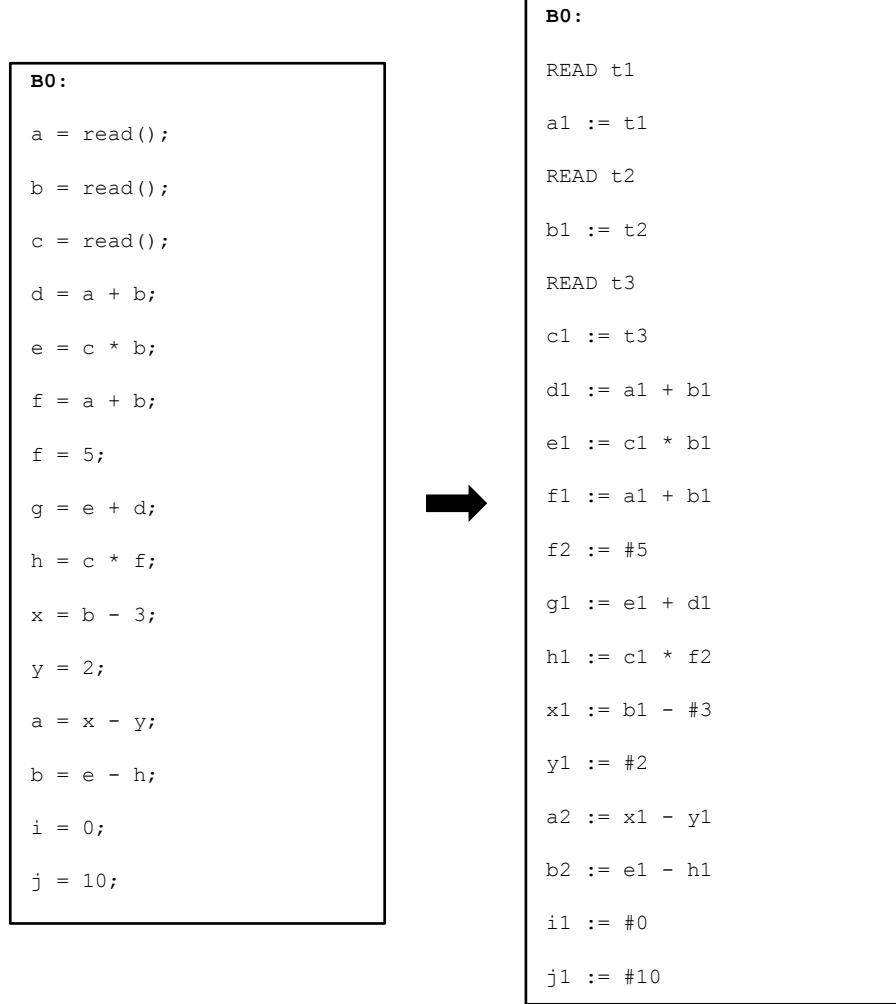


例如，对于上面这样的一段代码，在使用简单常量传播进行分析时，依次分析 B1、B2、B3 三个基本块，然而，我们使用工作表算法进行计算时，默认除了 Entry 之外的基本块都处于 unexecutable 状态，对于每次基本块跳转，先进行常量传播分析，将可跳转的基本块 B2 加入到工作表中，消除未被执行的基本块 B3。

3) 稀疏常量传播(sparse constant propagation)

要介绍稀疏常量传播，我们先需要介绍一系列概念。

稀疏常量传播中的“稀疏” (sparse)二字，指的是以**静态单一赋值形式 (static single-assignment form, SSA form)**的中间表示形式为基础，分析变量的 use-def 关系。静态单一赋值的中间代码，use-def 链是显式的，通过 SSA 形式的中间代码，我们可以轻易地通过单次遍历完成基本块内部的局部优化：公共子表达式消除、常量折叠、无用代码消除的实现。



以基本块 Bo 为例，例如，对于变量 a，其在基本块内部被定义了两次：a1 与 a2，我们将 a1 与 a2 看作不同的变量，对于每一个变量，均只被赋值了一次，那么，对于变量的赋值与使用关系，我们可以构造 def-use 表如下：

1 a1 := t1	a1 def: 1 use: 4 6	value: t1
2 b1 := t2	b1 def: 2 use: 4 5 6	value: t2
3 c1 := t3	c1 def: 3 use: 5 9	value: t3
4 d1 := a1 + b1	d1 def: 4 use: 8	value: a1 + b1
5 e1 := c1 * b1	e1 def: 5 use: 8 13	value: c1 * b1
6 f1 := a1 + b1	f1 def: 6 use:	value: a1 + b1
7 f2 := #5	f2 def: 7 use: 9	value: 5
8 g1 := e1 + d1	g1 def: 8 use:	value: e1 + d1
9 h1 := c1 * f2	h1 def: 9 use: 13	value: c1 * f2
10 x1 := b1 - #3	x1 def: 10 use: 12	value: b1 - #3
11 y1 := #2	y1 def: 11 use: 12	value: 2
12 a2 := x1 - y1	a2 def: 12	value: x1 - y1 = b1 - 5
13 b2 := e1 - h1	b2 def: 13	value: e1 - h1 = c1 * (b1 - 5)
14 i1 := #0	i1 def: 14	value: 0
15 j1 := #10	j1 def: 15	value: 10

常量折叠：因每一变量仅被定义一次，因此若对变量的赋值为一常量，可直接使用常量替换该定值，如变量 f2 等。

公共子表达式消除：对于公共子表达式 a_1+b_1 ，变量 a_1 与 b_1 在第 4 行与第 6 行同时被 use，因此可以进行公共子表达式消除。同时，记录变量的值，然后使用代数恒等式变换，能够暴露 b_1-5 这一公共子表达式，并予以消除。

无用代码消除：在基本块出口不活跃的变量，若在基本块的内部仅被 def 而未有 use，可认为其是无用代码并予以消除，如 $f1 := a_1 + b_1$ 等。

SSA 形式的中间代码，给优化带来了极大的便利，但是，构造 SSA 形式的中间代码的算法较为复杂，我们介绍一种基于支配树(dominator tree)的 SSA 形式的中间代码生成算法。

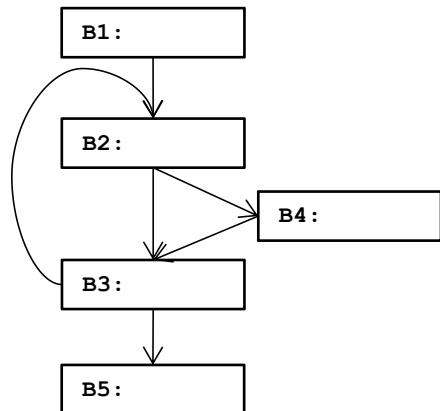
支配节点：对于某节点 n ，如果从入口节点到 n 的每一条路径都必须经过节点 m ，则节点 m 支配节点 n ，记为 $m \text{ dom } n$ ，每一个节点都支配自己。

支配关系满足自反性、反对称性和传递性：

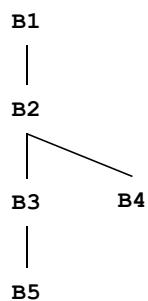
自反性： $a \text{ dom } a$

反对称性： $a \text{ dom } b, b \text{ dom } a \Rightarrow a = b$

传递性： $a \text{ dom } b, b \text{ dom } c \Rightarrow a \text{ dom } c$

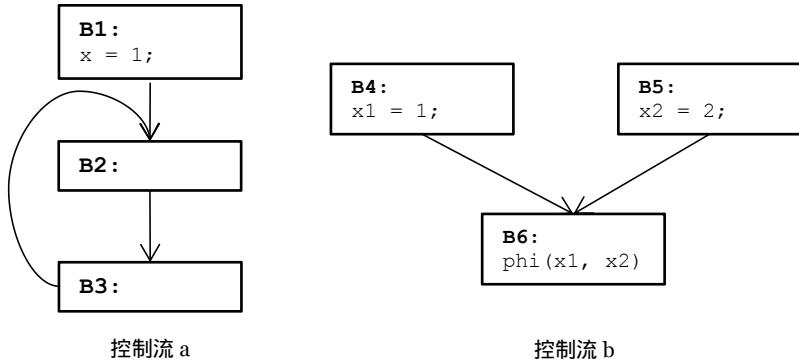


例如，在上面这个控制流图中，包含这些支配关系：B1 支配 B1、B2、B3、B4、B5，B2 支配 B2、B3、B4、B5，B3 支配 B3 和 B5 等。构造的支配树如下：



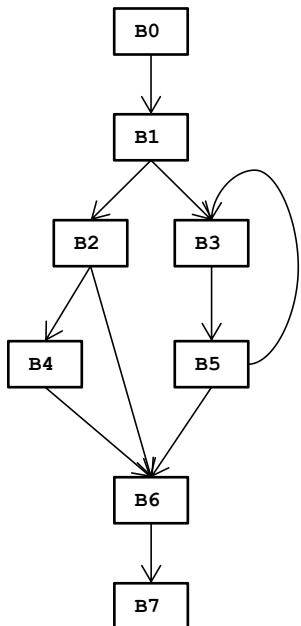
基于支配树与支配关系，能够进行基于区域的分析、进行循环相关的优化，也可以帮助生成 SSA 形式的中间代码。

为了生成 SSA 形式的中间代码，我们需基于支配树计算基本块的**支配边界 (dominance frontier)**，在支配边界上放置 phi 函数，进行程序状态的计算。



控制流 a 中，对变量 x 的赋值只存在于 B1 中，B1 支配 B2 与 B3，因此在 B2 中不需要添加 phi 函数。在控制流 b 中，赋值存在于 B4 与 B5 中，B4 与 B5 均不支配 B6，因此需添加 phi 函数。

计算控制流图支配边界的算法较为复杂，我们将其分为以下的步骤：



1) 计算支配关系

基于迭代算法，我们可以计算基本块之间的支配关系。在上文中，我们能够得知，一个基本块 m 支配另一个基本块 n，这表示对于所有由入口到 n 的路径均经过 m，因

此，对于基本块 n ，若 m 支配其的所有前驱基本块，则 m 支配 n 。因此，不同路径上的程序状态合并，与数据流分析模式中采用的 must 分析类似。

于是，我们构造支配情况更新函数如下：

$$D(n) = \cap_{p \text{ 是 } n \text{ 的一个前驱}} D(p) \cup \{n\}, D(n) \text{ 为支配 } n \text{ 的基本块}$$

根据支配的定义，支配节点 n 的基本块满足两个条件之一：

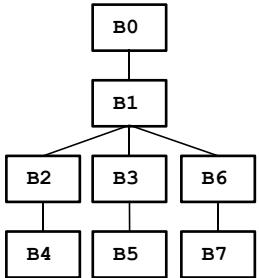
1. 支配其所有前驱基本块 $\cap_{p \text{ 是 } n \text{ 的一个前驱}} D(p)$
2. 是 n 本身 $\{n\}$

我们将支配情况初始化，然后进行支配情况的迭代更新。

block	D(n)
B0	{B0}
B1	{B0, B1, B2, B3, B4, B5, B6, B7}
B2	{B0, B1, B2, B3, B4, B5, B6, B7}
B3	{B0, B1, B2, B3, B4, B5, B6, B7}
B4	{B0, B1, B2, B3, B4, B5, B6, B7}
B5	{B0, B1, B2, B3, B4, B5, B6, B7}
B6	{B0, B1, B2, B3, B4, B5, B6, B7}
B7	{B0, B1, B2, B3, B4, B5, B6, B7}

block	D(n)
B0	{B0}
B1	{B0, B1}
B2	{B0, B1, B2}
B3	{B0, B1, B3}
B4	{B0, B1, B2, B4}
B5	{B0, B1, B3, B5}
B6	{B0, B1, B6}
B7	{B0, B1, B6, B7}

基于支配关系，我们构造支配树如下：

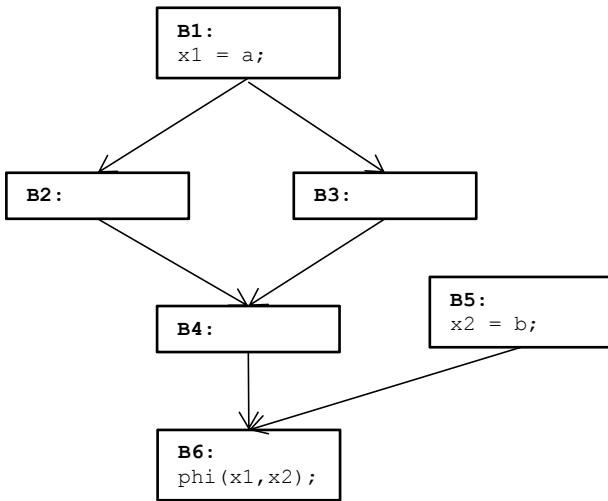


我们引入严格支配(strictly dominated)的概念，记作 $m \text{ sdom } n$ ，其中 $n \neq m$ ，基于支配树，我们可获得严格支配关系如下，其中 $\text{sdom}(n)$ 代表基本块 n 严格支配的基本块。

block	sdom(n)
B0	{B1, B2, B3, B4, B5, B6, B7}
B1	{B2, B3, B4, B5, B6, B7}
B2	{B4}
B3	{B5}
B4	{}
B5	{}
B6	{B7}
B7	{}

block	succ(dom(n))
B0	{B1, B2, B3, B4, B5, B6, B7}
B1	{B2, B3, B4, B5, B6, B7}
B2	{B4, B6}
B3	{B3, B5, B6}
B4	{B6}
B5	{B3, B6}
B6	{B7}
B7	{}

那么，基本块 n 的支配边界可以用以下的例子来解释。



程序内部对变量 x 的赋值共有两处：基本块 B1 与 B5。基本块 B1 支配基本块 B2、B3 和 B4，而到达 B6 不必经过基本块 B1，因此 B6 为 B1 的支配边界，同理 B6 也是 B5 的支配边界，因此在基本块 B6 内添加 phi 函数。

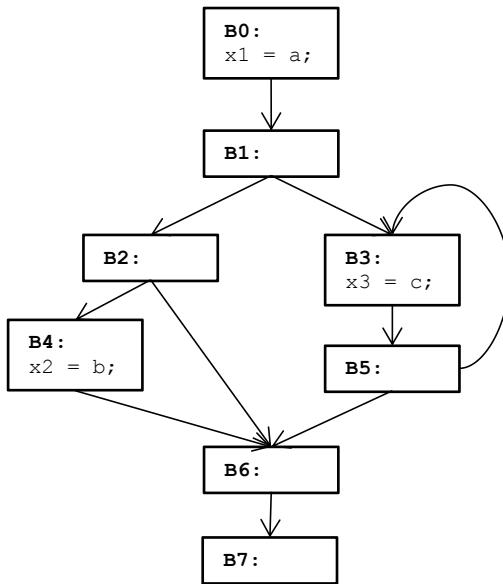
要寻找支配边界，即寻找基本块 n 的后继中，第一个不被基本块 n 支配的基本块，要寻找具备这种性质的基本块，可以从被支配的基本块的后继基本块中寻找。对于以上的这段程序，B1 支配 B1、B2、B3、B4，而 B1 的后继 B2、B3，B2 和 B3 的后继 B4，B4 的后继 B6 组成的后继集合{B2、B3、B4、B6}中，只有 B6 不被基本块 B1 支配，因此 B6 为 B1 的支配边界。

于是，我们用 $\text{succ}(\text{dom}(n))$ 表示基本块 n 支配的基本块的后继，如上文的表中所示。对于这些后继集合，我们只要从中去除基本块 n 支配的基本块，即可计算出 n 的支配边界。

block	$\text{succ}(\text{dom}(n))$
B0	{ }
B1	{ }
B2	{B6}
B3	{B3, B6}
B4	{B6}
B5	{B3, B6}
B6	{ }
B7	{ }

计算出支配边界之后，我们就可基于支配边界进行 phi 函数的添加。

假设之前的程序代码如下：



对变量 x 的赋值共有三处：Bo、B3、B4，其中 Bo 的支配边界集合为 $\{\}$ ，B3 的支配边界集合为 $\{B3, B6\}$ ，B4 的支配边界集合为 $\{B6\}$ 。我们使用工作表算法，进行 phi 语句的植入。

工作表初始化为 $w : \{Bo, B3, B4\}$

1. 从 w 中取得基本块 Bo，Bo 的支配边界集合为 $\{\}$ ，不添加 phi 语句

$w : \{B3, B4\}$

2. 从 w 中取得基本块 B3，B3 的支配边界集合为 $\{B3, B6\}$ ，在基本块 B3 和 B6 中添加 phi 语句。因基本块 B3 已添加 phi 语句，因此将 B6 加入到工作表 w 中。

$w : \{B4, B6\}$

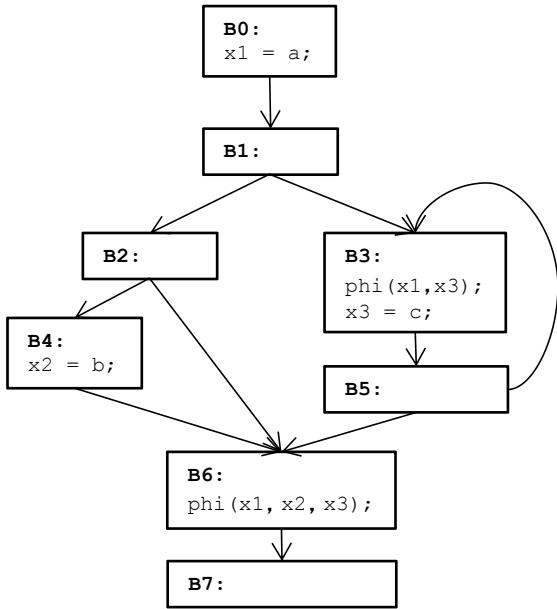
3. 从 w 中取得基本块 B4，B4 的支配边界集合为 $\{B6\}$ ，因基本块 B6 已添加 phi 语句，因此不做操作。

$w : \{B6\}$

4. 从 w 中取得基本块 B6，B6 的支配边界集合为 $\{\}$ ，因此不做操作。

算法结束。

因此，算法需要在基本块 B3 和 B6 中添加 phi 语句，并且将 $x1, x2, x3$ 添加到 phi 语句中，获得的程序控制流图如下所示：



添加了 phi 语句之后，便可构造 SSA 形式的中间代码。走了许久，不要忘了为什么出发，我们回看稀疏常量传播。

稀疏常量传播中，变量有 use 和 def 的关系。以上面的代码为例，变量 x_1 的 def 在 B_0 中，而存在 $\text{phi}(x_1, x_3)$ 、 $\text{phi}(x_1, x_2, x_3)$ 两处 use。相对于普通常量传播中在每个基本块的入口与出口进行变量状态的计算，在稀疏常量传播中仅需要在 def 和 use 处计算变量的状态，简化了大量的计算，这便是稀疏常量传播中“稀疏”二字的含义。

例如，假设 $x_4 = \text{phi}(x_1, x_2, x_3)$ ，则 x_4 的取值集合为 $\{a, b, c\}$ ，状态为常量 $a \cup$ 常量 $b \cup$ 常量 $c = \text{NAC}$ ，而不需要在基本块入口和出口反复进行计算。

4) 稀疏条件常量传播(sparse conditional constant propagation)

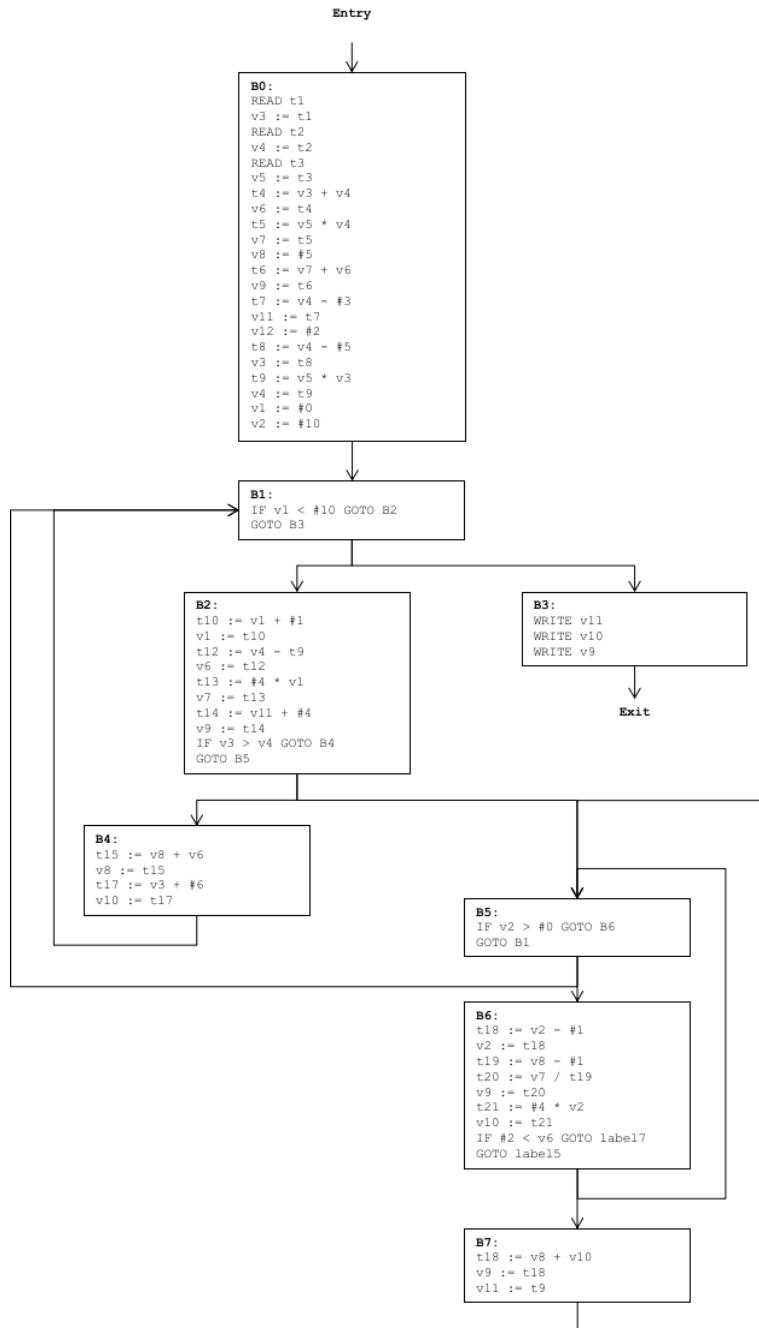
将稀疏常量传播与条件常量传播相结合，就是当前主流的常量传播算法：稀疏条件常量传播算法，通常简写为 SCCP，过程间稀疏条件常量传播则简写为 IPSCCP。

实验五只要求你基于简单常量传播算法进行常量传播，你也可以构造其他的常量传播算法，或将中间代码转化为 SSA 形式，并进行下一步的优化工作。

全局优化2 公共子表达式消除：

在实验五中，只要求你能够使用前文所述的可用表达式模式进行公共子表达式消除，因此，我们基于简单常量传播后的优化结果，运用可用表达式模式进行公共子表达式的消除。

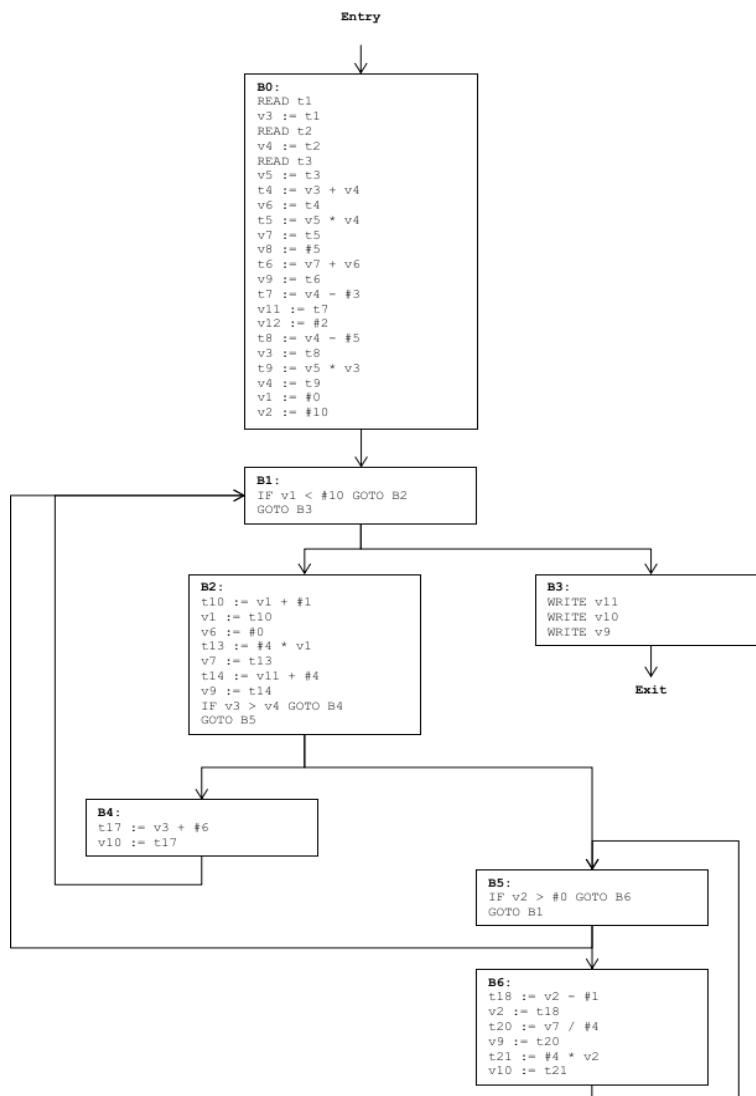
可用表达式模式已在前文叙述，此处不再赘述，优化后的程序应如下所示：



在经过子表达式消除之后，或许你可以发现，对于基本块B2中的语句 $t_{12} := v_4 - t_9$ 和 $v_6 := t_{12}$ ，经过对后续语句的扫描（或使用课本上所述的复制传播），变量 v_4 和 v_6 的值未被改变，而 $v_4 - t_9$ 的值为一定值 o ，此处可以使用常量 o 替换变量的使用。

在你进行子表达式消除的时候，或许你会感觉到，自己使用的子表达式消除方法十分呆板，且只有所有的前驱基本块中均存在该子表达式，子表达式才可被消除。在课本上提供了另一种冗余消除的方法：**部分冗余消除 (Partial Redundancy Elimination, PRE)**，用于公共子表达式消除及循环不变代码外提。这部分包含在课本第九章有关于部分冗余消除的内容中，因此不再赘述。

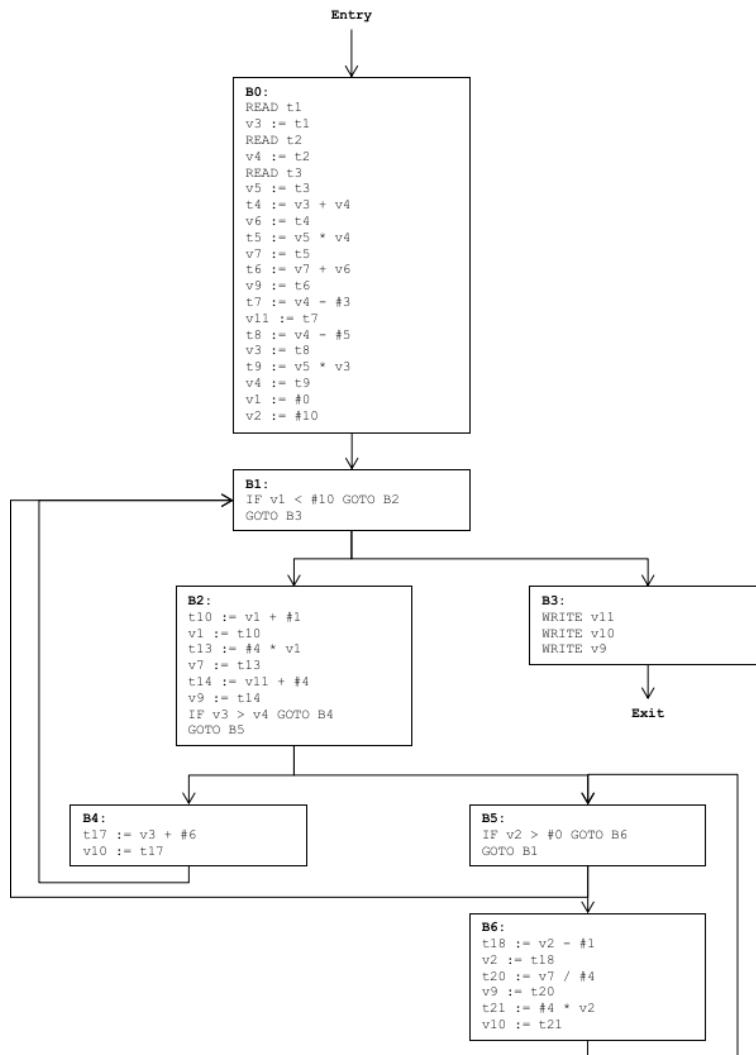
进行常量传播和子表达式消除之后，部分变量被常量替代，冗余表达式被消除或移动位置，因此需要重新进行常量折叠与控制流优化，去除不可达的基本块及代码（分支条件不可满足）。



全局优化3 无用代码消除：

全局无用代码消除关注被赋值但未被使用的变量，若对于变量的一次定义，在后续的程序中从未被使用，该定义可以被认为是无用的而被消除。通常，我们需要迭代式地删除程序内部的无用代码。在实验五中，要求你能够使用前文所述的活跃变量模式进行无用代码消除，当然，你也可基于生成的SSA形式的中间代码与def-use链，进行无用代码的消除。

活跃变量模式已在前文叙述，此处不再赘述，优化后的程序应如下所示：



全局优化4 循环不变代码外提：

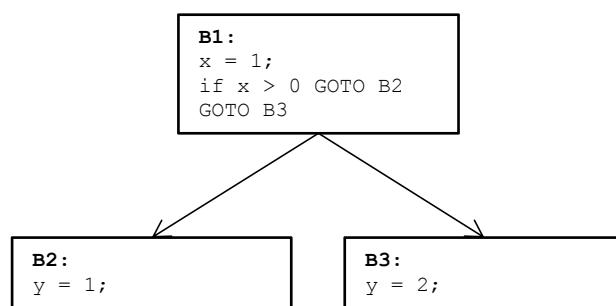
在中间代码优化中，针对循环的优化是一个重要的课题。循环内部的语句通常比其他语句执行的次数更多，如果我们仅把不必出现在循环内部的语句移动到外部，也能起到较好的优化效果。**循环不变代码外提(loop invariant code motion, LICM)**关注

那些每一次执行循环，得到的结果都不变的语句，对于这种语句，我们或许可以将他们移动到循环外部，从而完成优化过程。

在设计优化之前，我们首先要关注，有哪些语句被认为是可被移动的。你可能会觉得，循环不变的代码外提是一件十分简单的事情：对于一个赋值语句的RHS，构成该表达式的每一个变量，使用到达定值模式进行分析，如果这些变量都在循环外部被定义，那么这条语句就是可被移动的。

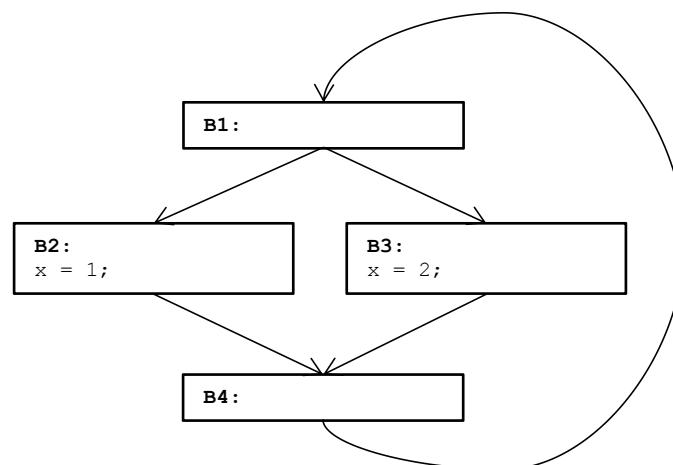
且不论你在算法中是如何定义“循环”的，可移动代码有很多需要关注的性质：

1. 被移动前是可达代码



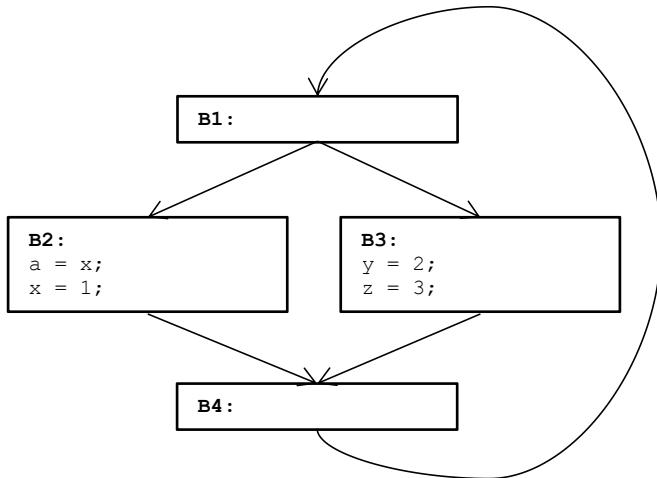
例如，对以上的这段代码，基本块B3中的代码均为不可达代码，如果在不可达代码消除之前将语句移动到其他位置，反而增加了冗余代码，更可能导致程序语义的改变。因此，在我们的实验指导中设计的优化管道，在循环不变代码外提和无用代码消除之前，设计了常量折叠和控制流优化，进行不可达代码的消除。如果你选择只完成这一部分的优化，你也可设计一些简单的全局扫描与常量折叠算法，进行不可达代码的判断与消除。

2. 循环内部仅存在一条对该变量的赋值语句



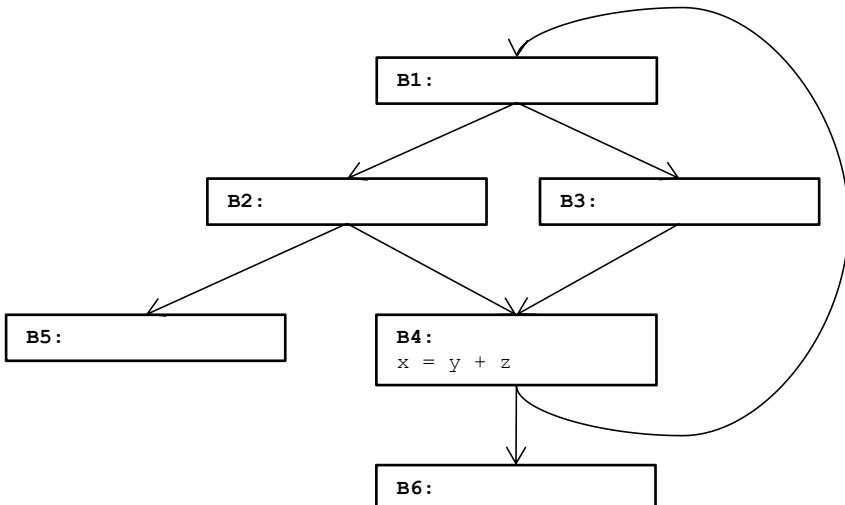
对于这段代码，循环内部存在两条对x的赋值语句，无论将哪一条外提都会对语义产生影响。

3. 对变量进行def之前，循环内部不存在use



如基本块中对x的赋值，在赋值之前存在对x的使用，若将其简单地外提到循环的外部，则会改变x的取值，从而改变语义。

4. 变量定义的基本块能够支配所有的循环出口



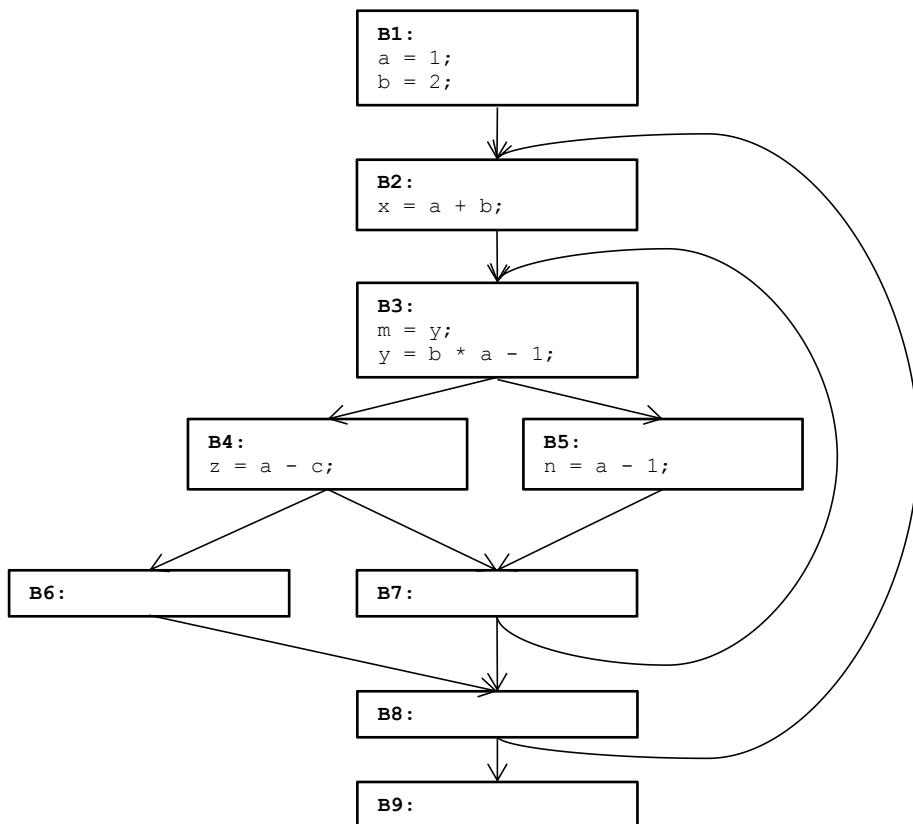
例如，对于这段代码，循环存在两个出口基本块B2与B4，虽然变量y与z均在循环的外部被定义，但是该语句并不能被移动到B1或更前的基本块（但该语句或许可被移动到基本块B6）。在实验五中，我们只要求你将代码移动到基本块入口、入口的前驱基本块或在入口之前新构造一个基本块中，而如果你的算法足够智能，也可将其移动到后继的基本块中。

因此，可移动的代码需要满足以下的特性：

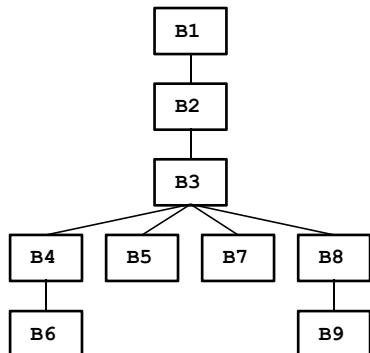
- 循环不变
- 所处的基本块能够支配所有的出口基本块
- 循环内部不存在其他对该变量的赋值
- 所处的基本块能够支配所有存在该变量使用语句的基本块

对于支配的相关概念，请翻阅本书常量传播中有关稀疏常量传播的部分。

同时，循环不变代码外提中，我们通常都是从最内层的循环进行代码外提，对于最内层循环的查找与计算，请参考龙书 9.6 节的内容，以下是一个由里到外进行循环不变代码外提的例子：



我们首先构造循环代码的支配树：



以上的这段代码，包含了两个循环：{B3,B4,B5,B7}和{B2,B3,B4,B5,B6,B7,B8}，我们将循环{B3,B4,B5,B7}标记为 L1，{B2,B3,B4,B5,B6,B7,B8}标记为 L2，循环 L1 存在两个“出口” B4 与 B7，循环 L2 仅存在一个“出口” B8。

我们基于从里到外的顺序，先处理内层循环 L1：

1. 循环 L1 内部能够支配所有出口的基本块仅有 B3，其中存在的语句：

$m = y;$
 $y = b * a - 1;$

基本块 B4、B5 中的语句不为可被外提的语句。

2. 循环 L1 内部不存在其他对变量的赋值：

变量 y, m 在循环 L1 中均不存在其他的赋值。

3. 循环 L1 内部赋值语句产生的变量的 def 关系能够支配所有对该变量的 use：

对变量 y 的赋值因不能支配所有的 use，故不能作为可被外提的语句。

4. 对赋值语句的 RHS 中存在的变量使用到达定值模式进行分析：

$m = y$ 中变量 y 的值非定值，因此也不可以作为循环不变的代码外提。

经过 4 个步骤，循环 L1 处理完毕，然后我们处理循环 L2：

1. 循环 L2 内部能够支配所有出口的基本块有 B2 和 B3，其中存在的语句：

$x = a + b;$
 $m = y;$
 $y = b * a - 1;$

2. 循环 L2 内部不存在其他对变量的赋值：

变量 x, y, m 在循环 L2 中均不存在其他的赋值。

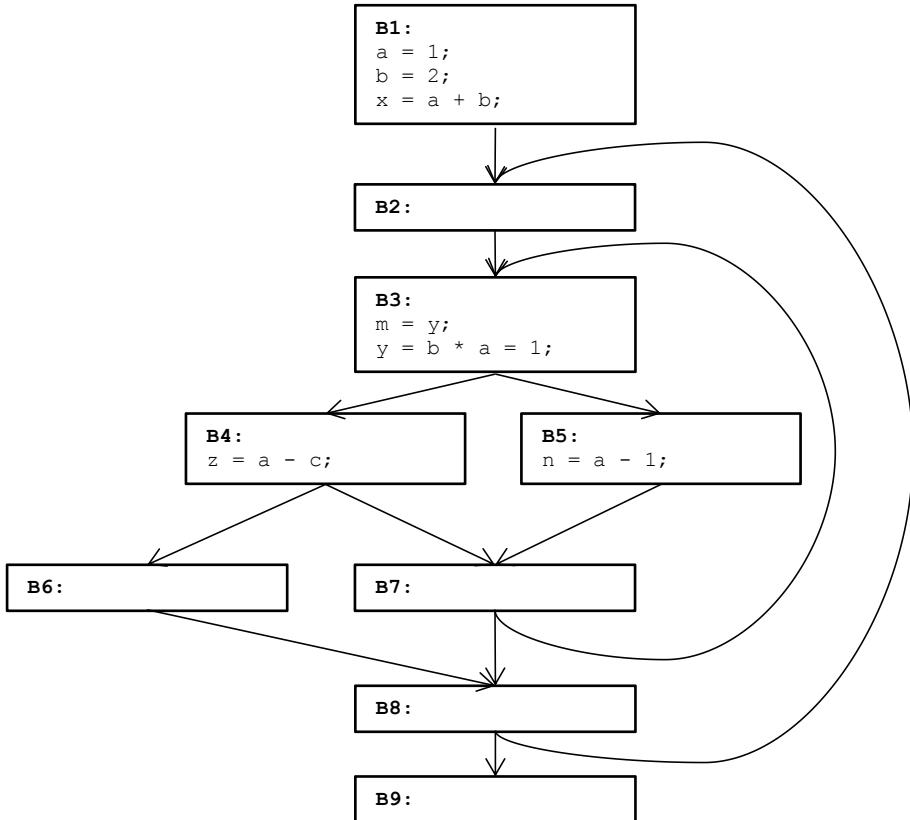
3. 循环 L2 内部赋值语句产生的变量的 def 关系能够支配所有对该变量的 use：

基本块 B3 中的变量已被分析过，而变量 x 在循环中不存在 use，因此满足条件。

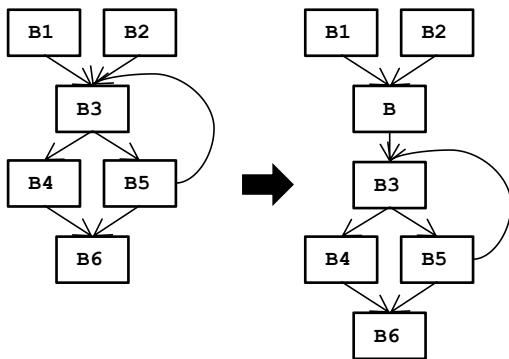
4. 对赋值语句的 RHS 中存在的变量使用到达定值模式进行分析：

语句 $x = a + b$ 中，RHS 中存在的变量 a 与 b ，使用到达定值模式分析后，其值均为定值，因此可作为循环不变的代码外提。

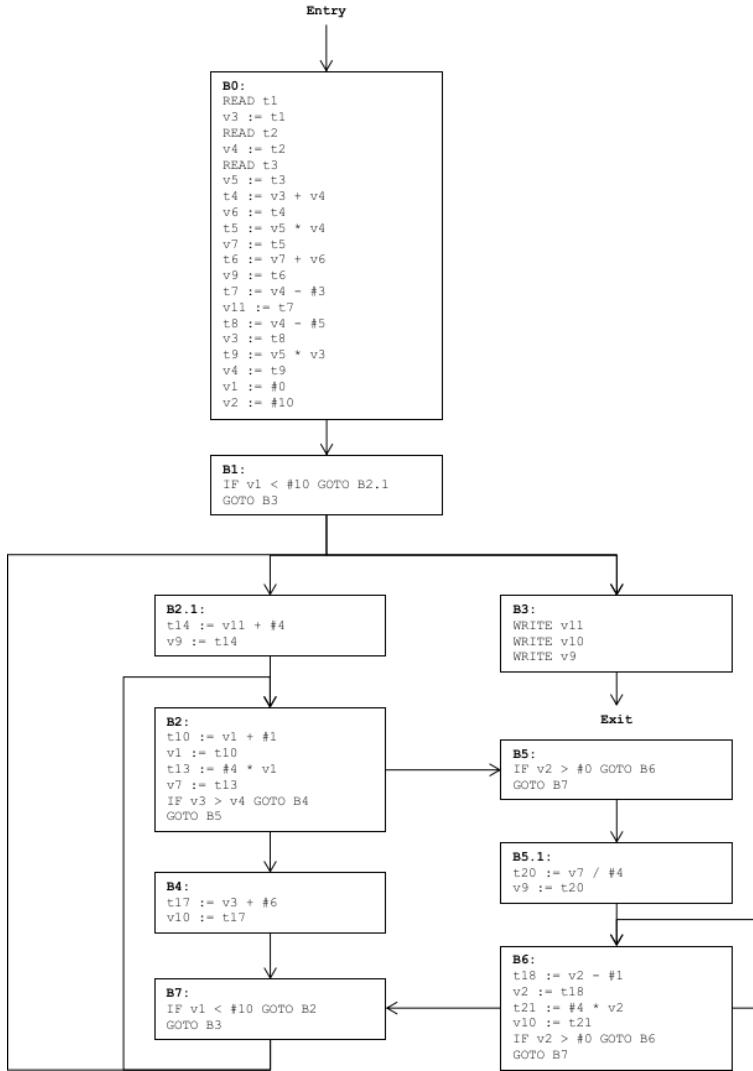
对循环的分析结束后，构成的控制流图如下：



我们将语句 $x = a + b$ 从基本块 B2 移除，将其移动到循环“入口”的前驱基本块 B1 中。有时，循环的入口存在多个前驱基本块，此时我们可额外增加一个基本块，并将语句置于其中：



在实验五中，你需要基于到达定值模式及上述可移动代码的相关性质，迭代式地进行代码移动，生成的代码可以是这样的：



相较于前三种优化，与循环相关的优化工作量较大，所需实现的算法也较多，存在一定难度。

全局优化5 归纳变量强度削减：

实验五中另一个循环相关的优化是归纳变量强度削减(Induction Variables Strength Reduction)。在优化管道的设计中，归纳变量强度削减通常跟随在循环不变代码外提之后。

基础归纳变量(basic induction variable)是在循环内部每次赋值都加或减一个常数c，并且该变量不能被替换成常数，且不是循环不变的变量，如：

$$x = x + c \text{ 或 } x = x - c, \text{ } c \text{ 为一常量}$$

则，**归纳变量(induction variable)**可能是：

- 一个基础归纳变量x

- 在循环中仅有一条针对该变量的赋值语句，且赋值语句的RHS为一个基础归纳变量的线性方程组，形如 $y = x * c_1 + c_2$ ，其中 c_1 与 c_2 为常量， x 为一基础归纳变量。

构成基础归纳变量a的家族被定义为：一个变量构成的集合A，对于A中的任意变量b，在循环内部对b的赋值语句均为基础归纳变量a的线性方程组。

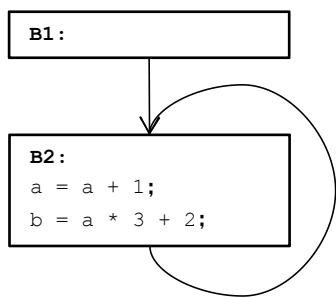
我们首先要关注，有哪些变量可被看作能够进行强度削减的变量。

对于基础归纳变量a的家族集合中的一个变量b，b能够被表示为：

$$b = a \text{ op } c_1 + c_2$$

其中op表示任何二元运算

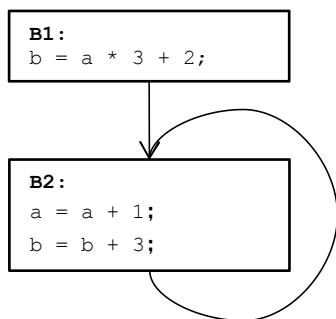
当程序内部存在如以下的语句：



我们可以使用 b' 代替b，使用代数恒等式变化将代码转变为：

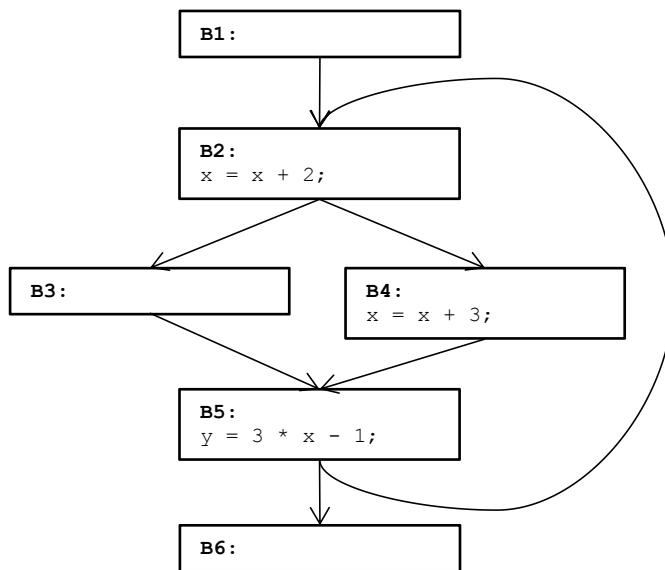
$$b' = b' + 3 * 1;$$

则代码将转化为：



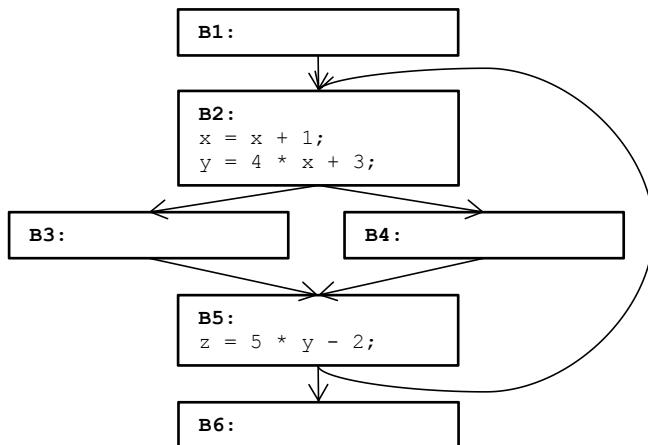
对于可进行强度削减的变量，同样存在一些特性：

1. 对变量的赋值能够支配所有的使用（与循环不变代码外提相同）
2. 对于基础归纳变量a与其家族的成员b，不存在循环外部的a对b的赋值产生影响。
3. 循环内部不存在形如 $a = a + c$ (c 为常量) 之外对a的赋值语句，且对基础归纳变量的赋值语句能够支配所有对家族内部变量的赋值语句。

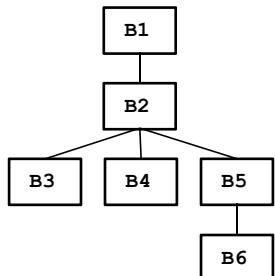


例如，对于以上的这段代码，B4不支配B5，因而不能进行归纳变量强度削减。

以下是一个归纳变量强度削减的例子：



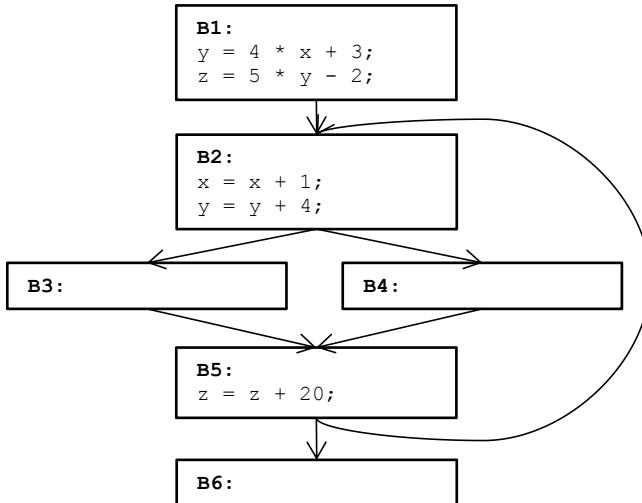
1. 对于以上的代码，构造支配树（见稀疏常量传播中支配树相关）：



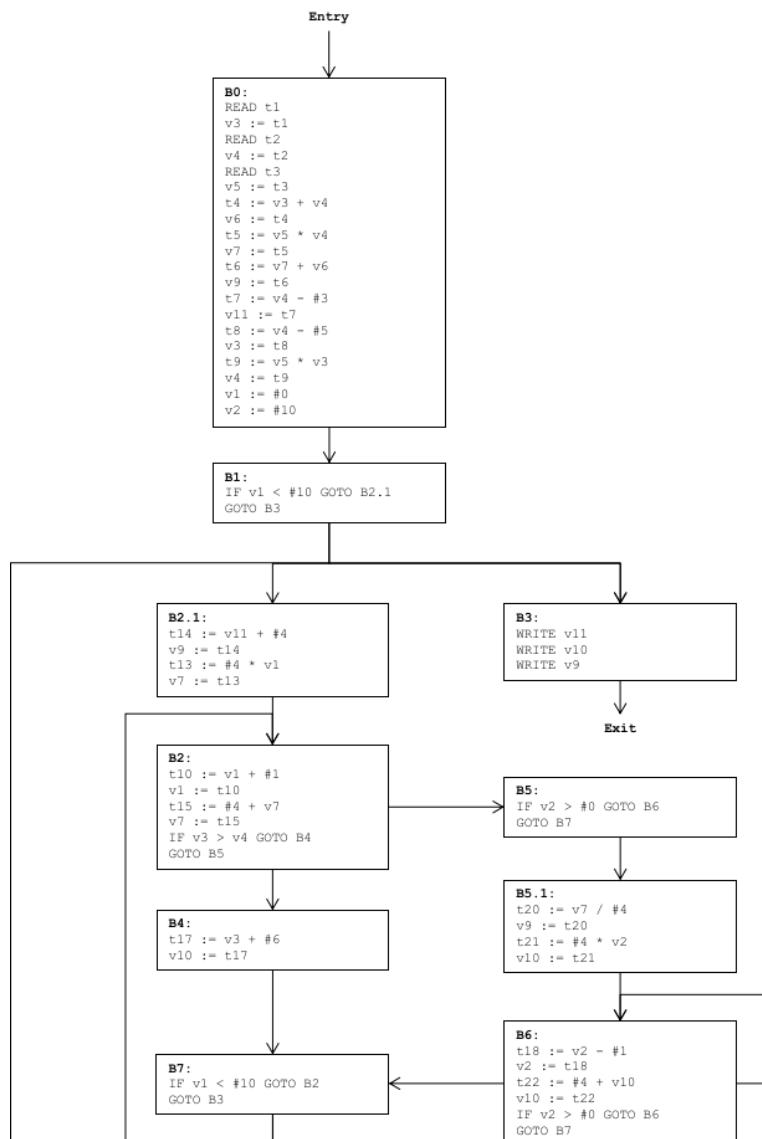
2. 基于支配树，寻找基础归纳变量，寻找到变量x。

3. 迭代地构造基础归纳变量x的家族。

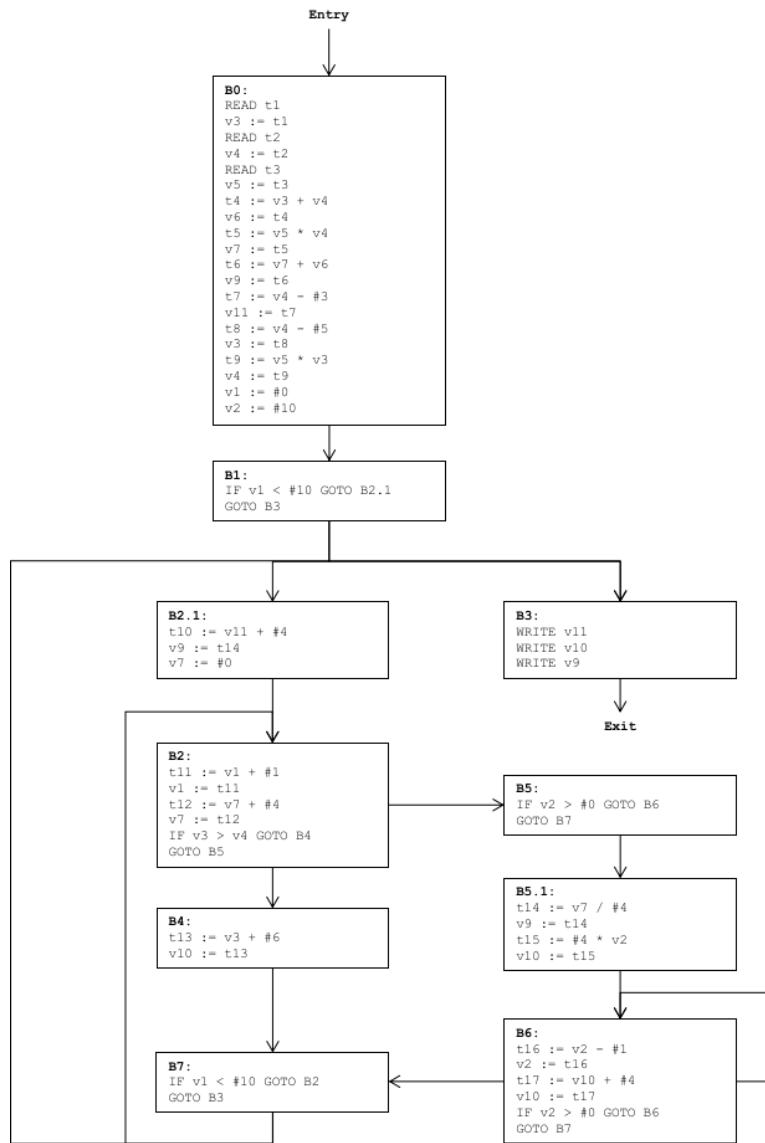
4. 使用代数恒等式进行替换，实现强度削减：



在实验五中，你需要基于到达定值算法及上述归纳变量的相关性质，迭代式地进行强度削减，生成的代码可以是这样的：



基于生成的基本块，进行控制流优化与值的重新标号，最终生成的优化后的中间代码如下：



在实验五中，你可选择以上的五个全局优化的任意组合，构造优化管道并实现优化算法。你需要使得优化后的中间代码在执行时经过尽可能少且开销较少的语句，并且确保生成的中间代码的语义不发生改变。

6.2.9 过程间优化

至今为止，我们所提到的优化都为 **过程内优化（Intraprocedural optimization）**，过程内优化只关注单个函数内部的程序优化，而相较于前文提到的局部优化（基本块内部）与全局优化（函数内部），**过程间优化（Interprocedural**

optimization) 将优化范围扩大到多个函数，关注数据在调用者与被调用者之间的流动。

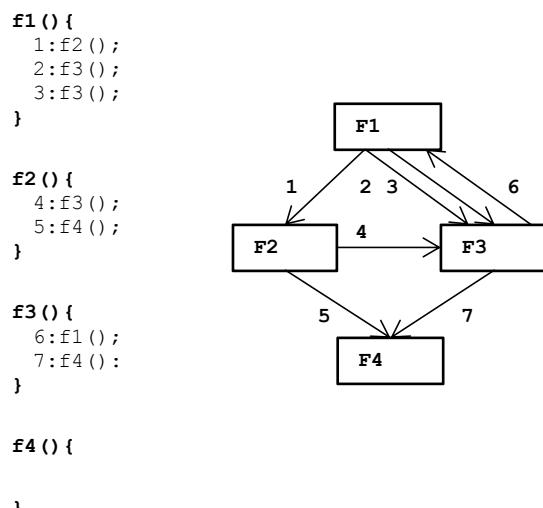
部分优化，如指针相关的优化，必须基于过程间优化进行。出于安全性的考虑，我们认为一次函数调用可能改变任何可被访问的指针变量所指向的内容，如果仅使用过程内分析进行程序优化，对指针相关的优化将极其保守，且十分低效。

```
int func (int x)
{
    void *p = (void*) x;
    int r = (int)p;
    return r;
}
```

c语言中，int类型的值可轻易与指针类型互转，若使用过程内分析的方式分析函数调用后的程序状态，会导致指针分析结果的全面失效。

调用图 (Call Graph) :

进行过程间优化，我们首先需要了解，程序内部在哪些地方进行函数调用、调用者和被调用者分别是哪个函数及哪些数据在调用之间流动。为此，我们使用一个与过程内调用所使用的控制流图类似的图来表示调用之间的关系：调用图。

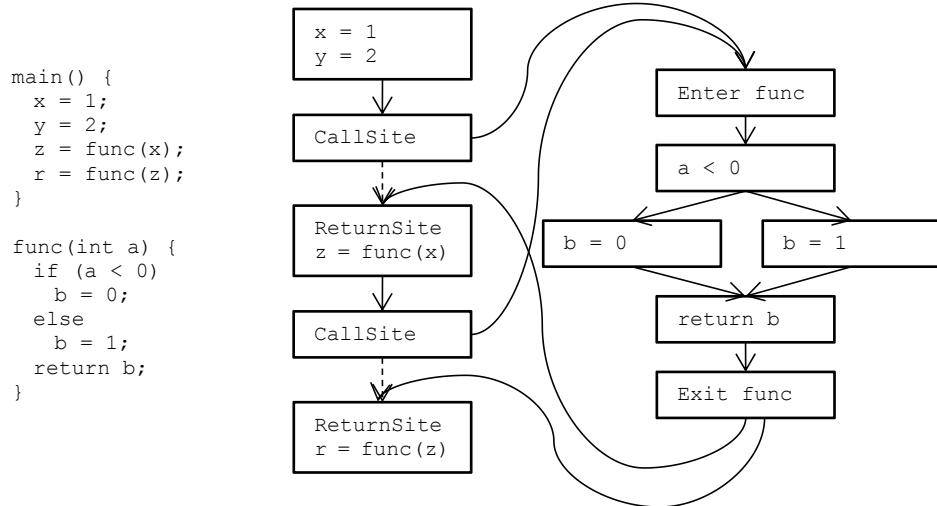


例如，对于以上这段代码，共有七个函数调用，构造的函数调用图如右所示。

有了调用图，我们就可以开始从事过程间的分析与优化。

函数内联 (Inlining) :

进行过程间的分析，首先能被想到的，自然是将函数的调用与返回加入到一个大的控制流图中，以“过程内”的方式进行过程间的分析。



在进行分析时，我们将被调用的函数内联进来：

```
main() {
    x = 1;
    y = 2;
    if (x < 0)
        b1 = 0;
    else
        b1 = 1;
    z = b1;
    if (z < 0)
        b2 = 0;
    else
        b2 = 1;
    z = b2;
}
```

这样的分析方式简单，易于理解，也是进行过程间分析的主要方式之一。然而，这种方法存在着开销昂贵的问题，且对于部分场景难以使用。

高昂的开销：

```
main() {
    while(a > 0){
        func1();
    }
}

func1(){
    func2();
}

....
```

函数中存在着大量循环与函数的嵌套调用，使用内联进行分析时会使得代码规模指数性增长，进行分析与优化时开销极大。

难以使用的场景：

```
func1(x) {  
    ...  
    func1(x-1);  
}
```

函数的递归调用，难以使用函数内联的方式进行分析。

在进行过程间优化时，可以使用函数内联，将跨函数的优化转变为过程内的优化，并基于先前所述的方法进行分析与优化。

函数摘要 (Summary) :

基于摘要的过程间分析是另一种进行分析的方式。在进行过程内的优化时，我们可以同步收集函数的各个输入参数从函数入口到出口之间的状态改变。对于不同的优化需求，我们构造相对应的抽象与内存模型，然后基于数据流分析或值流分析的方式，构造函数入口的程序状态 S_{Entry} 与函数出口的程序状态 S_{Exit} 之间的状态映射。

```
func(int a, int b) {  
    x = 1;  
    y = 2;  
    a = a + b - y;  
    b = x + y - 1;  
    a = 3 * a - b;  
    return a;  
}  
  
a  
↓  
3 * (a + b) - 8  
b  
↓  
2
```

```
main() {  
    z = func1(x,y); → main() {  
    }  
    z = 3 * (x + y) - 8;  
}
```

在进行优化时，采用由内到外的顺序，依次基于 $\langle S_{Entry}, S_{Exit} \rangle$ 的映射关系，进行程序状态的更新与计算，进而实现程序的优化工作。

过程间优化广泛用于指针相关优化、Java等面向对象的语言对虚方法的调用优化和并行代码的优化等，现有的过程间优化基于可达性分析、值流分析等理论，虽然在实验五中没有要求，但是学有余力的同学可自行查阅资料进行进一步的学习与理解。