中间代码优化

2025

南京大学计算机学院

课程实践目标

- 实践目标

主要内容：实验内容是为一个小型的类C语言（C--）实现一个编译器。如果你顺利完成了本实验任务，那么不仅你的编程能力将会得到大幅提高，而且你最终会得到一个比较完整的、能将C--源代码转换成MIPS汇编代码的编译器，所得到的汇编代码可以在SPIM Simulator上运行。

课程实践总共分为五个阶段：词法和语法分析、语义分析、中间代码生成、目标代码生成以及中间代码优化。每个阶段的输出是下一个阶段的输入，后一个阶段总是在前一个阶段的基础上完成。其中，目标代码生成以及中间代码优化均基于第三次中间代码生成。

中间代码优化

- 有哪些程序片段需要被优化?
  - 未能执行到的代码
  - 未被使用的赋值
  - 恒为一常量的值
  - ……
- 应当运用哪些优化方法?
  - 数据流分析算法等
- 应当以什么顺序进行代码优化?
  - 优化管道的设计

中间代码优化

- 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通常按顺序执行。

- 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、无用代码消除、循环相关优化等。

- 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（针对无用代码削除等）。

实验内容

- 目标

实验五的任务是在词法分析、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。

实验五是实验三（中间代码生成）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。

实验参考框架：https://github.com/BryanHeBY/cmmc_optimizer

实验内容

- 实验内容（必做项）

对于给定的程序，在实验三生成的IR 基础上，对IR 进行中间代码优化。

假设：同实验三中间代码生成部分，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，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。其中**局部优化**为必做内容，包括**局部公共子表达式优化、局部无用代码消除优化、局部常量折叠优化。**同时优化模块还需包含控制流图简化，**去除空基本块**，优化执行顺序。

实验内容

• 输入格式

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。

实验内容

• 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。

评分标准：

1. 优化后的结果与原始语义相同
2. 优化后的代码行
3. 优化后单次执行的指令数量

实验内容

- 实验内容（可选内容1）

  完成基于数据流分析模式的优化：

  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

实验内容

- 实验内容（可选内容2）

完成循环不变代码外提优化，将每次执行结果都不变的表达式，移动到循环外部。

实验内容

- 实验内容（可选内容3）

完成归纳变量强度削减优化，将循环内部与归纳变量相关的高代价运算转换为低代价的运算。

基本块与控制流图

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。

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。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。

基本块内部的优化与跨基本块的优化分别被称为局部优化和全局优化。

数据流分析

进行数据流分析时，我们关注：

1. 在一条语句执行的前后，程序状态发生的变化；
2. 在不同执行路径的交汇处，程序状态的合并应采取什么样的方式。

一个数据流分析框架(D, V, R, F)由下列元素所组成：

（1）一个数据流方向D，它的取值包括前向（Forward）和后向（Backward）。**到达定值**与**可用表达式分析**是前向分析，**活跃变量分析**是后向分析。

（2）一个半格(V, R)，V代表程序状态的集合，R是集合的交运算或并运算，用于表示在基本块入口处对不同的前驱（或在出口处对不同的后继）的程序状态的合并。

（3）一个从V到V的传递函数族F，用于刻画基本块内部每条语句对程序状态造成的变化。

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。

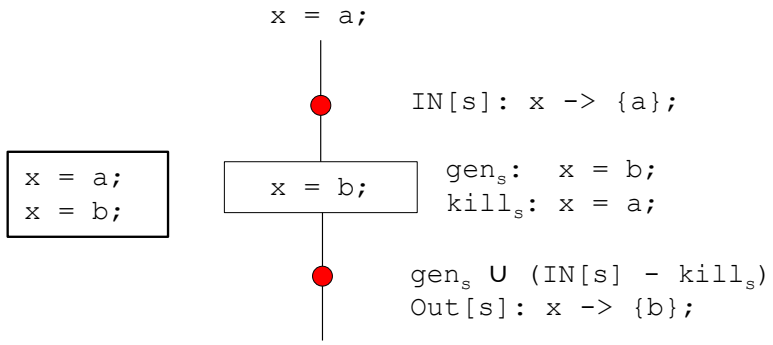
传递函数：

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

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，当我们想知道变量可能的赋值集合的变化的情况，我们构造的传递函数可以是这样的：

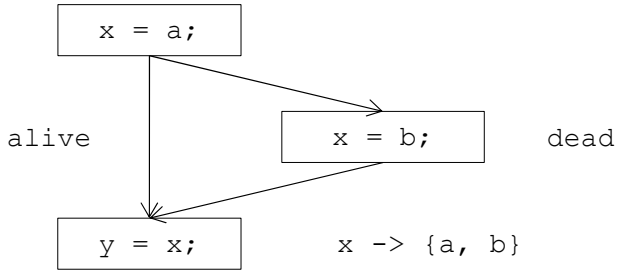
$$OUT[s] = gen_s \cup (IN[s] - kill_s)$$

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。

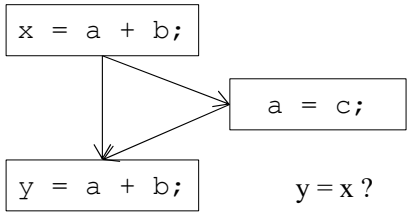


May分析与Must分析

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，我们想要知道对于变量v的一次定义definition d，在程序点p上是否依然可能是存活的，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，对于经过程序点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再次赋值，那么在该程序点上表达式才是可用的。



局部优化

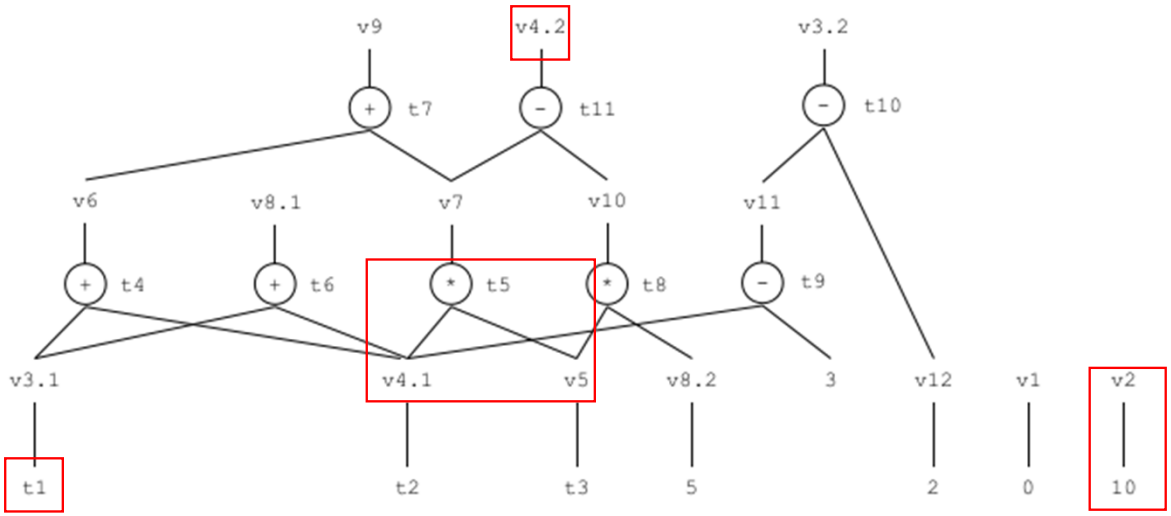
考虑如下的中间代码（基本块B0）

```
B0:
a = read();
b = read();
c = read();
d = a + b;
e = c * b;
f = 5;
g = e + d;
h = c * f;
x = b - 3;
y = 2;
a = x - y;
b = e - h;
i = 0;
j = 10;
```

```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
t6 := v3 + v4
v8 := t6
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

采用龙书8.5节的方法进行局部代码优化，将中间代码转化为DAG

局部优化




```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
t6 := v3 + v4
v8 := t6
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

我们由下而上地分析有向无环图中的节点：

底层节点表示常量或是定值；

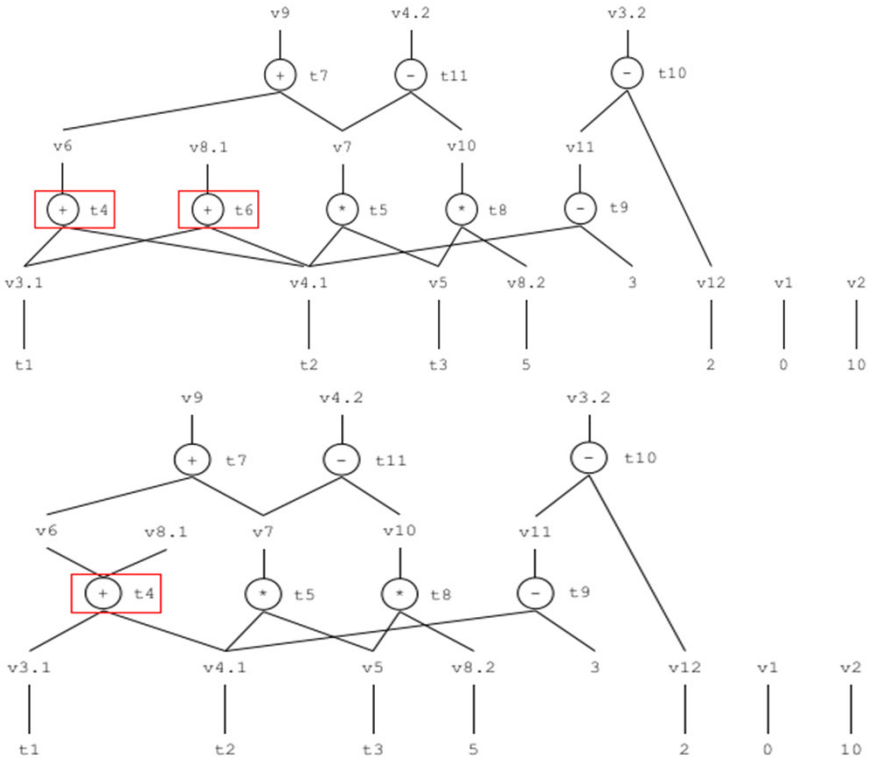
顶层节点表示当前基本块内部赋值但未使用的变量；

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。

局部分析

公共子表达式消除(common subexperssion elimination)：

如果表达式E在某次出现之前已经被计算过，并且表达式中的操作数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。

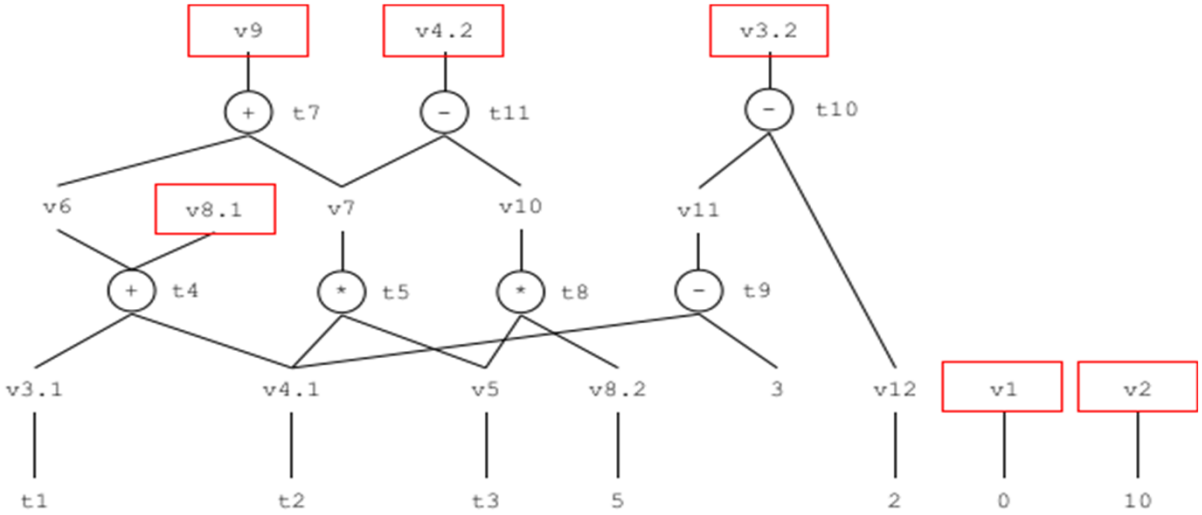



```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
t6 := v3 + v4
v8 := t6
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

局部分析

无用代码消除(dead code elimination):

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（或寄存器）、进行计算是没有必要的。

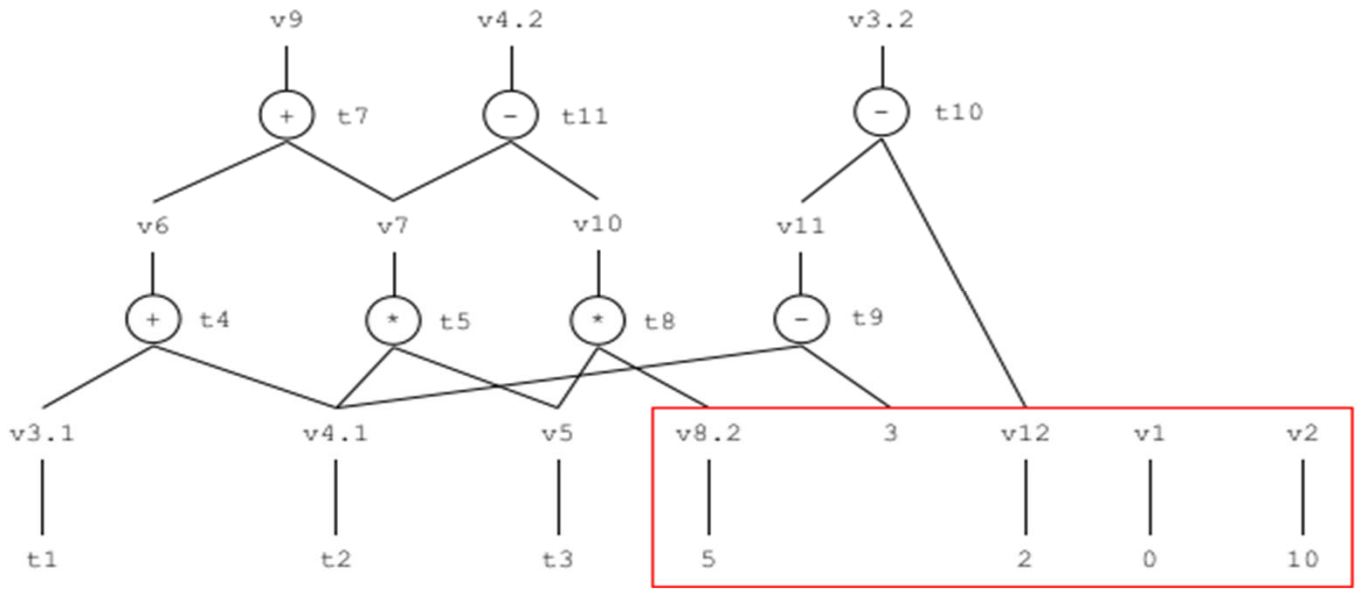



```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
v8 := t4
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

局部分析

常量折叠(constant folding):

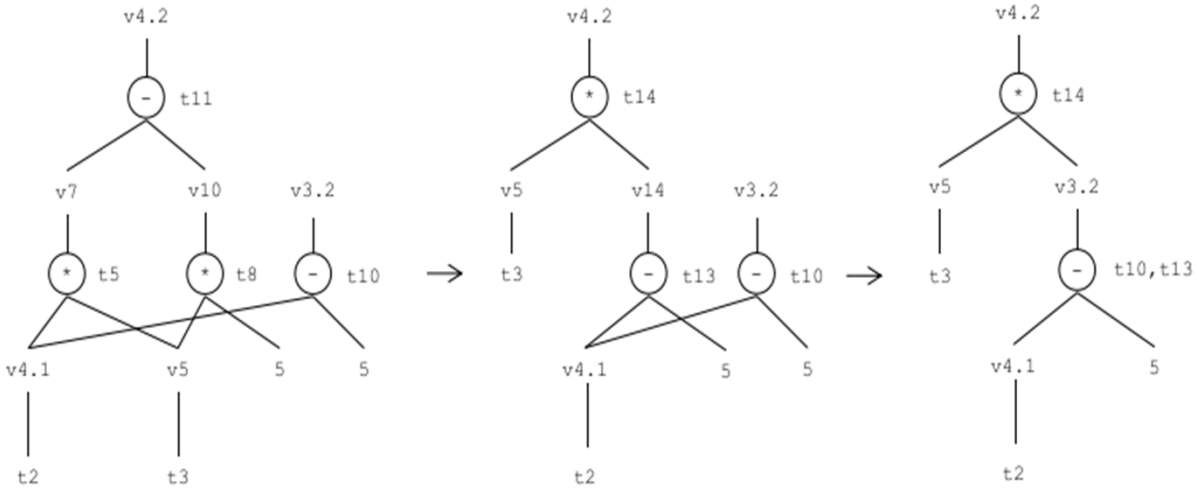
程序内部如v8.2 = 5， v12 = 2，v1 = 0，v2 = 10等将变量定义为一个常量的赋值语句，在该变量被再次定义之前，可将变量替换为一个常量，从而简化计算的过程。



局部分析

常量折叠(constant folding):

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。



v4.2 = v4.1 * v5 - v5 * 5
v3.2 = v4.1 - 5

=>
v4.2 = v5 * (v4.1 - 5)
v3.2 = v4.1 - 5

=>
v3.2 = v4.1 - 5
v4.2 = v3.2 * v5

局部优化

优化后的结果：

```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
t6 := v3 + v4
v8 := t6
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
v8 := #5
t6 := v7 + v6
v9 := t6
t7 := v4 - #3
v11 := t7
v12 := #2
t8 := v4 - #5
v3 := t8
t9 := v5 * v3
v4 := t9
v1 := #0
v2 := #10
```

局部优化

```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
t6 := v3 + v4
v8 := t6
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

将中间代码转化为SSA形式的中间代码后，
可以通过单次遍历完成三种优化：

```
B0:
READ t1
READ t2
READ t3
t4 := t1 + t2
t5 := t3 * t2
t6 := t1 + t2
t7 := #5
t8 := t5 + t4
t9 := t3 * t7
t10 := t2 - #3
t11 := #2
t12 := t10 -
t11
t13 := t5 - t9
t14 := #0
t15 := #10
```

➡

```
B0:
READ t1
READ t2
READ t3
t4 := t1 + t2        def: t4   use: t1,t2
t5 := t3 * t2        def: t5   use: t2,t3
~~t6 := t1 + t2~~
t7 := #5             def: t7
t8 := t5 + t4        def: t8   use: t4,t5
t9 := t3 * #5        def: t9   use: t3
t10 := t2 - #3       def: t10  use: t2
t11 := #2            def: t11
t12 := t2 - #5       def: t12  use: t2
t13 := t5 - t9
t14 := #0
t15 := #10
```

```
子表达式：
t4 := t1 + t2
t5 := t2 * t3
t8 := t4 + t5
t9 := t3 * #5
t10:= t2 - #3
t12:= t2 - #5
```

```
   t5 - t9
= t2 * t3 - t3 * 5
= t3 * (t2 - 5)
= t3 * t12
def: t13 use: t3,t12
```

例如，当程序遍历至t6 := t1 + t2时，查询子表达式表，可以发现给t4赋值的表达式中的操作数与运算符均相同，则t6与t4可相互代替。

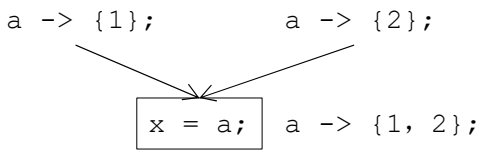
在遍历至t9 := t3 * #5时，能够得知t7的值为一定值#5，因此使用#5代替t7。

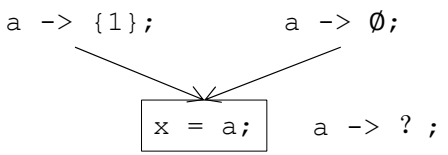
在遍历至t13 := t5 - t9时，通过查询当前存在的子表达式，能够进行恒等式变换。

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。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，描述了程序内部对变量v的定义definition d，能否到达程序点p，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。



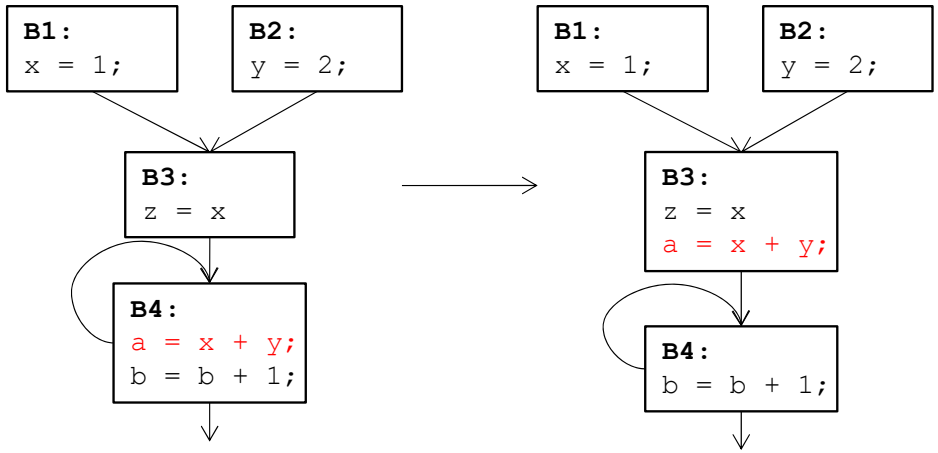


到达定值

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。



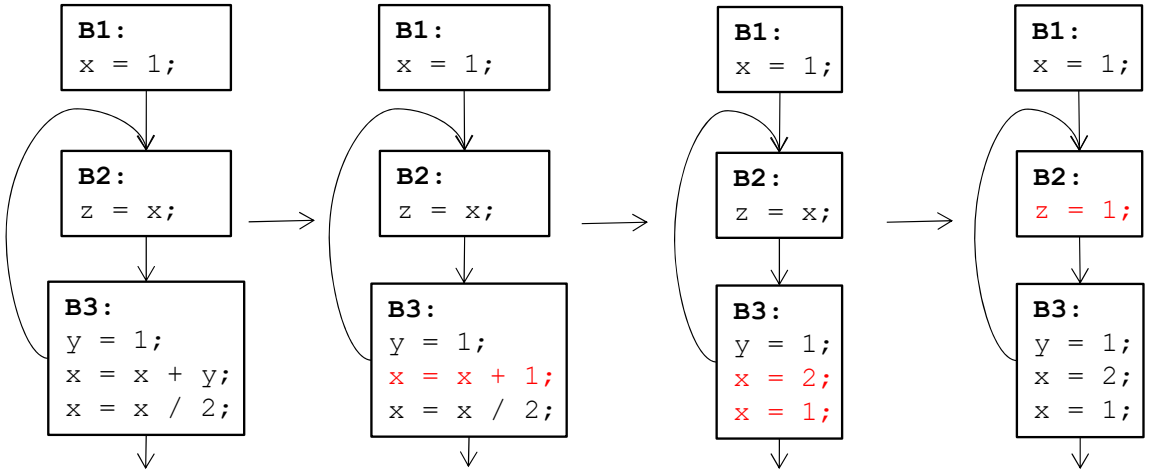
到达定值

到达定值分析的主要用途有以下几项：

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，使用到达定值进行分析，可以将其中的所有变量都替换为常量。



到达定值

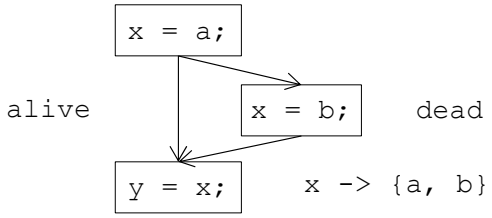
我们首先要选择，我们是应当使用may分析还是must分析进行程序状态的分析？

对于到达定值问题，分析的对象：

对于程序内部的变量v的一次定义definition d，在程序点p上是否可能还存活

即，只需要有一条从d出发到达p的路径，变量v未被重新定义，则在程序点p上v的值就还是d定义的。

为此，我们使用may分析的方式进行程序状态的分析。

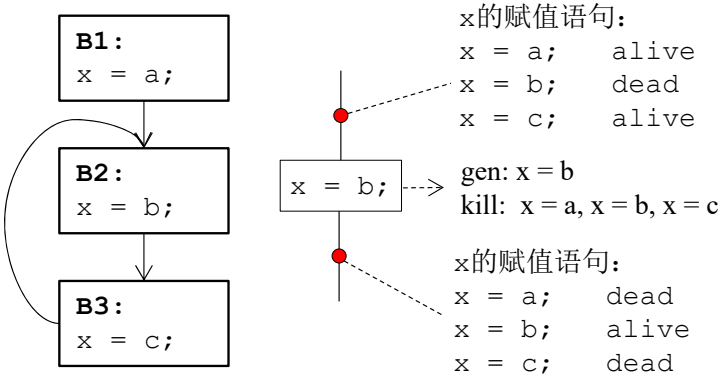


到达定值

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。

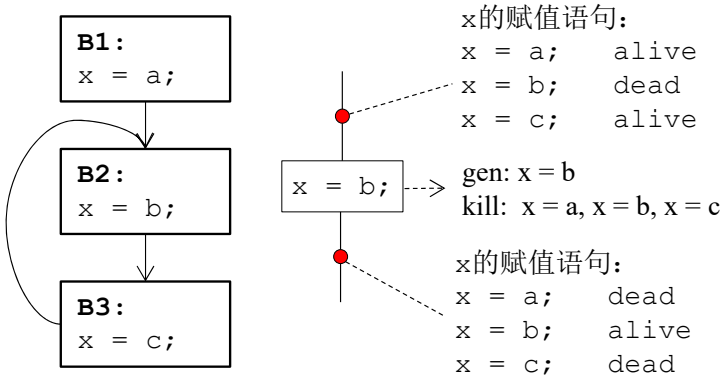
传递函数用于描述程序内部的状态变化，那么，对于到达定值问题，我们只需关注其中definition语句。程序内部通常存在多条对同一变量v赋值的语句，但是在某一条对变量v的赋值语句d: v = x执行完毕之后，v的值只可能是“定值”x。于是，对于变量v，其状态转换的传递函数定义为：

$$OUT[s] = gen_s \cup (In[s] - kill_s)$$



到达定值

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。



$$OUT[s] = gen_s \cup (In[s] - kill_s)$$

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，有两条赋值语句存活:x = a, x = 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 = b，“杀死”了其他所有对x的赋值情况。

那么传递函数中gen_s，In[s]，$kill_s$分别为：

gen_s：语句x = b中，对于变量x新的definition: x = b。

In[s]：执行语句之前存活的definition

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

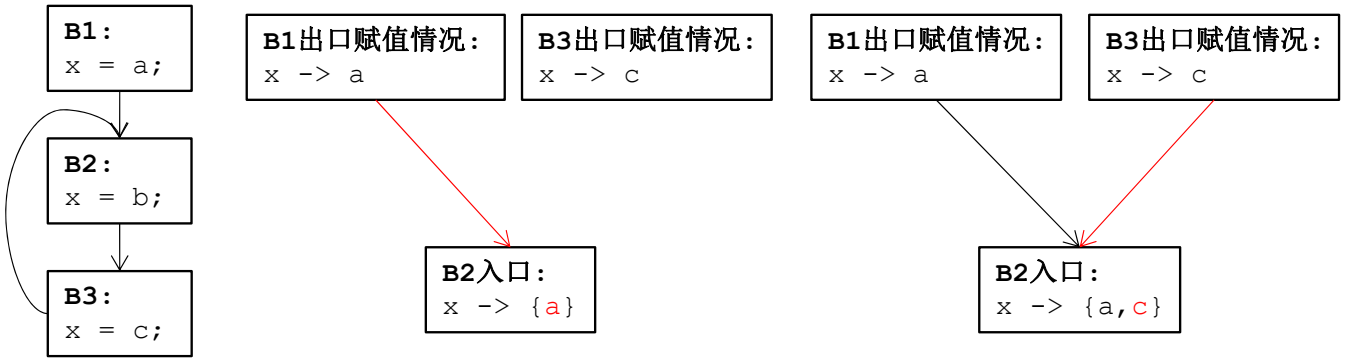
到达定值

要构造数据流分析算法，需要思考几个问题：

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的所有可能存活的definition。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下面的例子中，描述程序状态变化。



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的definition的并集。

到达定值

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。

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。

龙书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] = ∪P是B的一个前驱 OUT[P];
         OUT[B] = genB ∪ (In[B] - killB);
      }
   }
```

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。
为什么结束的条件是基本块的OUT都不改变？

到达定值

程序状态的有界性：

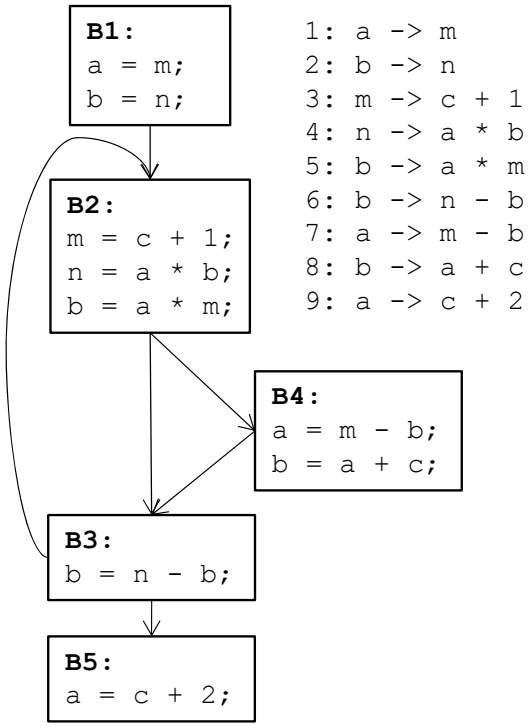
这段代码共有9条赋值语句，与赋值语句相对应的赋值情况也有9条，每一条赋值情况有两种可能状态：alive和dead，如果用一个一维数组来表示每个赋值情况的状态，用0代表dead，用1代表alive，那么(0,0,0,0,0,0,0,0,0)表示9条赋值情况均失效，(1,1,1,1,1,1,1,1,1)表示9条赋值情况均生效，每一个程序点上的状态，至多有2^9种情况，程序的状态数量是有界的。

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 -> 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，即在程序点p上，x -> 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（已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。



到达定值

工作列表算法：

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)，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];

$IN[B] = \cup_{P是B的一个前驱} OUT[P]$;

$OUT[B] = gen_B \cup (In[B] - kill_B)$;

if (OLD_OUT ≠ OUT[B])

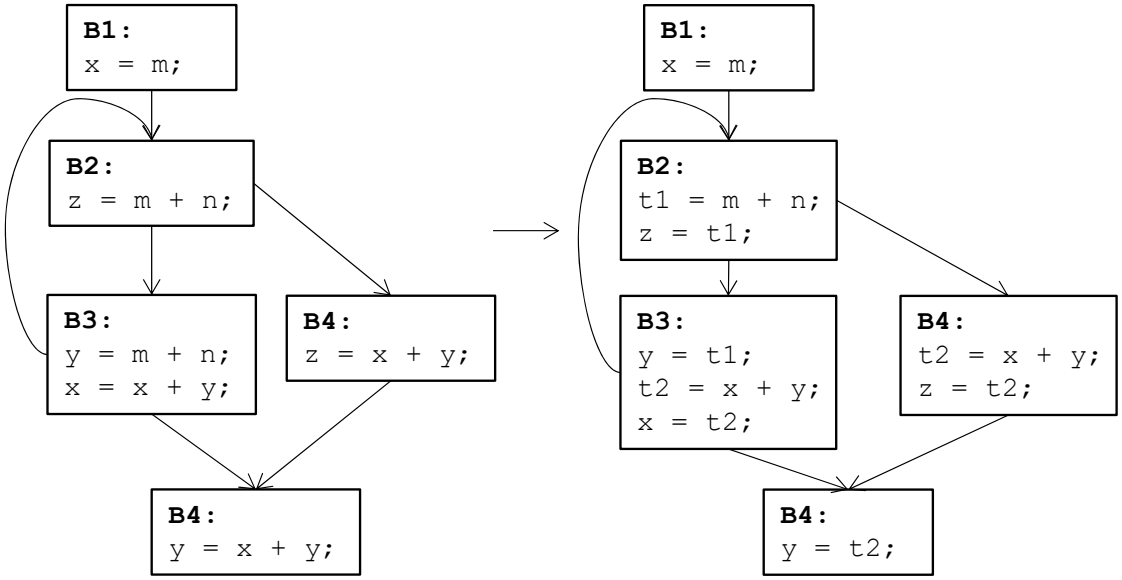
把基本块B的所有后继加入到工作表中

}

可用表达式

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

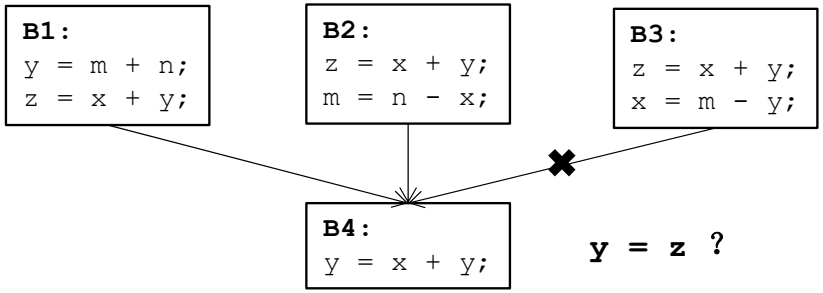


表达式m+n与x+y，在程序内部被多次计算，因此可以使用t1与t2代替其求值的结果，简化表达式的计算。

可用表达式

与到达定值相同，首先我们确定应当使用may分析还是must分析。

可用表达式问题，用于分析在某一程序点p上的表达式e，对于所有经过程序点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。



可用表达式

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。

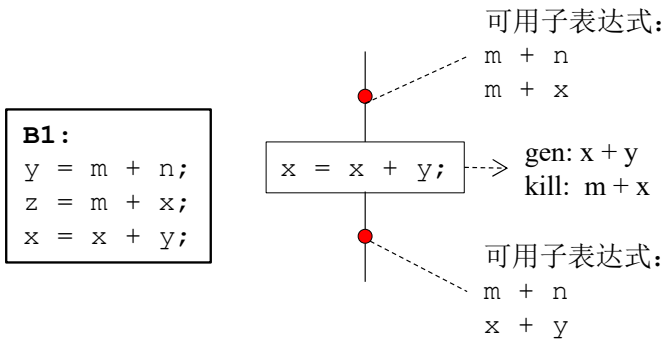
表达式b + c，构成表达式的变量有b和c两个，

s1: a = b + c，b + c生效，

s2: b = d，对于变量b进行重新赋值，使得表达式b + c的求值失效。

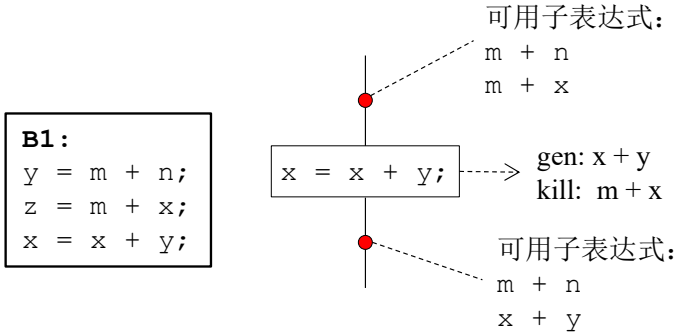
于是，对于表达式b + c，其状态转换的传递函数定义为：

$$OUT[s] = e\_gen_s \cup (In[s] - e\_kill_s)$$



可用表达式

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。



$$OUT[s] = e\_gen_s \cup (In[s] - e\_kill_s)$$

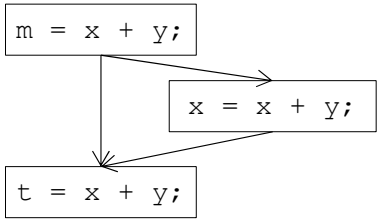
在执行语句x = x + y之前，程序内部的可用子表达式共有两个：m + n与m + x，语句x = x + y生成了子表达式x + y，同时对x重新进行赋值。因表达式m + x中存在对x的使用，对x的重新赋值使得m + x的求值失效。于是，我们认为，语句x = x + y，“生成”了可用子表达式x + y，“杀死”了可用子表达式m + x。

那么传递函数中e_gen_s，In[s]，e_kill_s分别为：

e_gen_s：语句x = x + y中，生成了可用子表达式x + y。

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。

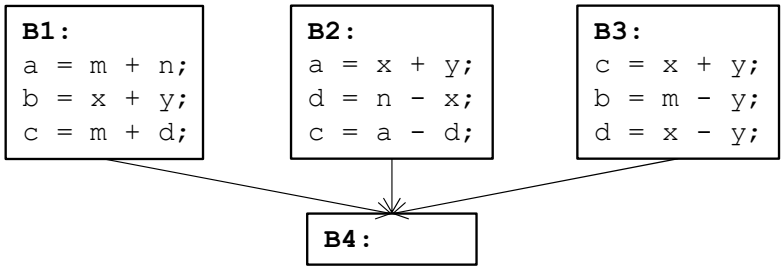


可用表达式

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：

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




```
B4_IN = B1_OUT ∩ B2_OUT ∩ B3_OUT
      = {m + n, x + y, m + d} ∩ {x + y, n - x, a - d} ∩
        {x + y, n - x, a - d}
      = {x + y}
```

可用表达式

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] = U;

while(某个OUT值发生了改变){

    for(除ENTRY之外的每个基本块B){

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

        OUT[B] = $e\_gen_s \cup (In[s] - e\_kill_s)$;

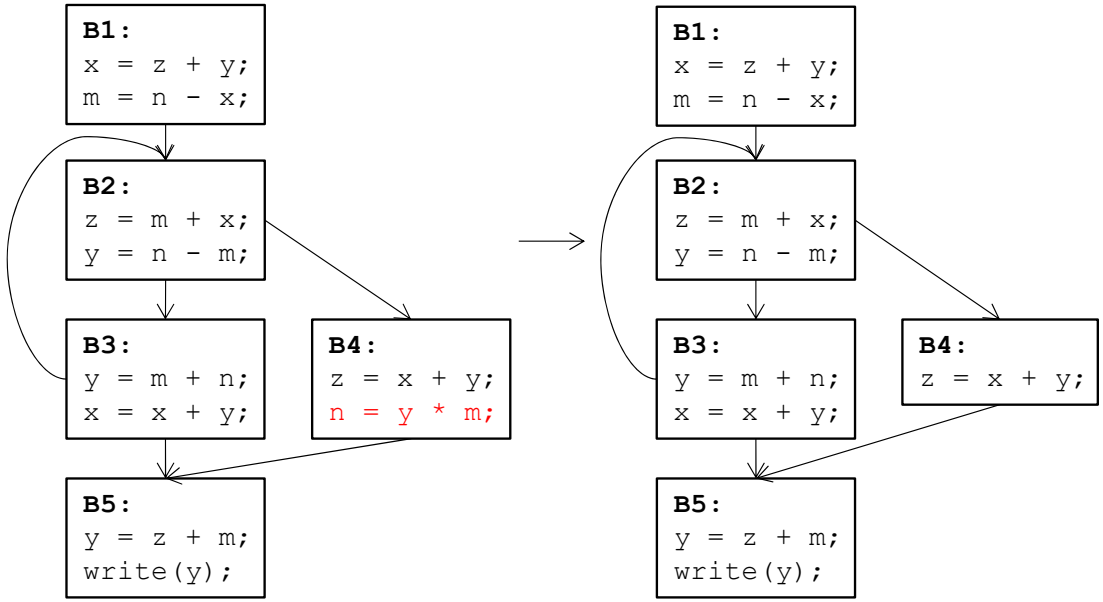
    }

}

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。

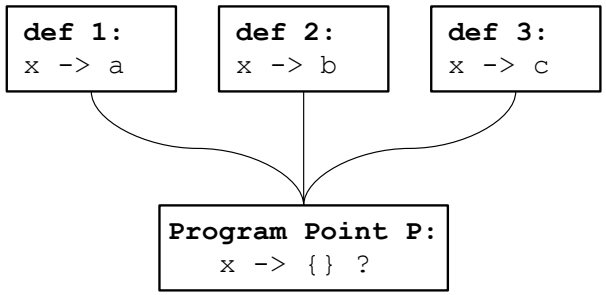
在程序优化的过程中，活跃变量通常用于进行**无用代码消除(dead code elimination)**。通过活跃变量模式对函数内部进行分析，消除未被使用的赋值与未执行的程序代码：



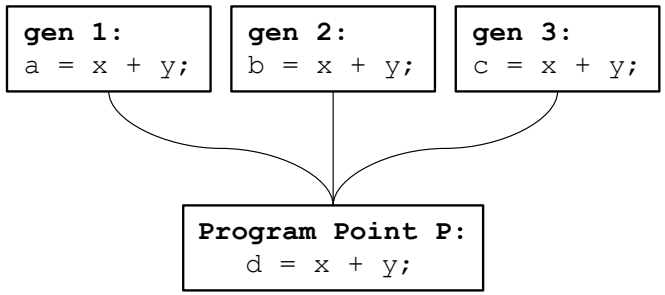
活跃变量

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前向遍历的遍历方式，而活跃变量分析要用后向的遍历进行程序状态的分析。

到达定值：
分析某一程序点p上对某一变量x的所有可能存在的definition，对应多条经过p的路径上前驱基本块中对变量x的definition语句。

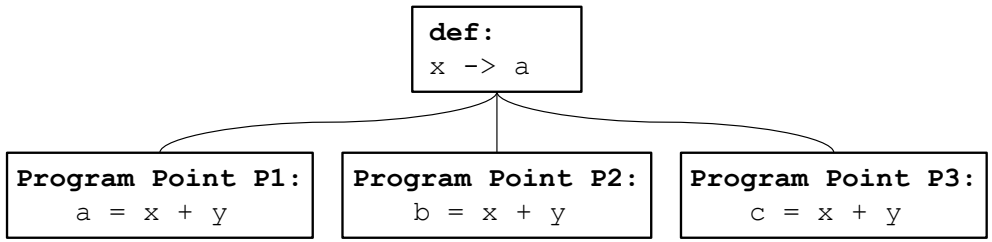


可用表达式：
分析在某一程序点p上是否存在可用的子表达式，子表达式在多条经过程序点p的前驱基本块内部被计算，算法分析在程序点p处子表达式是否还生效。

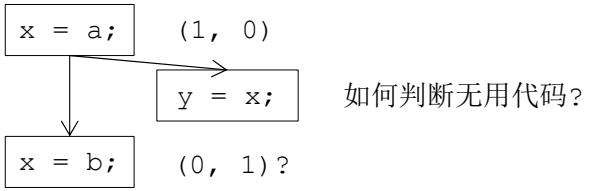


活跃变量

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：

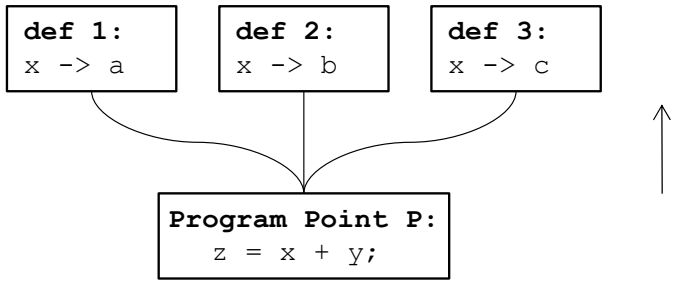


对变量x的一次定义def: x -> a，可能在后继基本块中的多个程序点p1、p2、p3等被使用，因此使用前向遍历较为复杂。



活跃变量

因此，我们使用后向遍历的方式进行数据流分析，将问题转化为如下的情况：



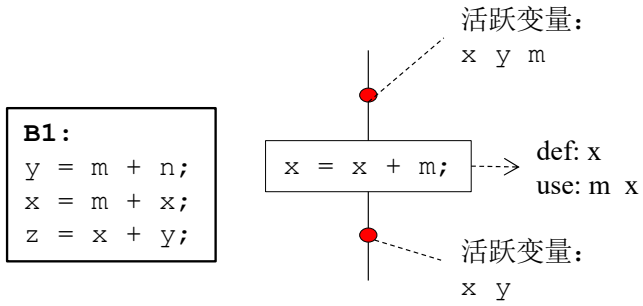
对于程序点p上的语句，对x进行了使用(use)，那么我们分析这一条语句使得哪些前驱基本块内部对于变量x的赋值(define)不是无用代码。我们使用后向遍历的方式，因此我们构造的传递函数与控制流约束函数较之前两种分析模式也有一些不同。

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。使用后向遍历时，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，匹配一组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：

$$IN[s] = use_s \cup (OUT[s] - def_s)$$

活跃变量

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：



$$IN[s] = use_s \cup (OUT[s] - def_s)$$

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use_s，OUT[s]，def_s分别为：

use_s：语句x = x + m中，使用了变量x与m。

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。

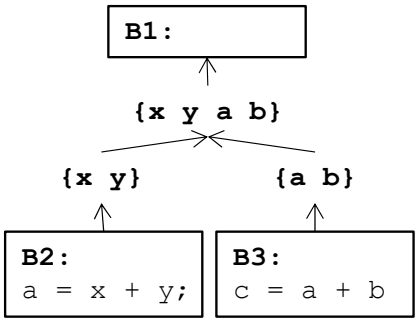
则传递函数$IN[s] = use_s \cup (OUT[s] - def_s)$的语义为，对于待分析的赋值语句s，执行语句之后存在的赋值情况记为Out[s]，若其对某变量x进行赋值，则先从赋值语句中去除该变量x $(OUT[s] - def_s)$，然后将当前语句使用的变量use_s加入到程序状态中。

活跃变量

对于控制流约束函数而言，活跃变量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，我们将控制流约束函数构造为如下的形式：

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

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。



B1的后继基本块有两个：基本块B2和B3，要分析在基本块B1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B2和B3分别提供了活跃变量x、y和a、b，因变量在后继基本块B2与B3中被使用，所以变量均可能在B1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。

活跃变量

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向分析的方式，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){

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

        $IN[B] = use_B \cup (OUT[B] - def_B)$;

    }

}

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，针对于生成的中间代码，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。（三地址码）

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于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较为接近，所不同的是，到达定值中对于变量的definition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.任意符合该变量类型的常量值, 值集为{c}

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值。{c1,c2,...}

3.UNDEF，表示未定义的值。表示我们尚未获得有关于变量的赋值相关信息。 {}

常量传播

z = x + y z -> ?

1. x与y中有至少一个变量状态为UNDEF，且没有状态为NAC的，这表示x与y中至少包含一个变量，在当前可能执行的路径上还未被定义，或我们对变量的赋值情况知道的信息太少（初始化状态）, 并且另一个值也未被定义或为一定值，那么，x+y的状态也应该是UNDEF的。

y\x	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	
c2	UNDEF		
NAC			

常量传播

2. x为一常量c1，y为一常量c2，那么z = x + y可转化为z = c1 + c2，z的状态为c3 = c1 + c2，为一常量。

y\x	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	
c2	UNDEF	c3	
NAC			

3. x和y中至少有一个状态为NAC，则z = x + y的取值集合也有多个值，因此z的状态为NAC。需注意的是，若变量x为UNDEF，y为NAC，虽然当前的分析未有对x的赋值，但是由于y已有多个可能取值，所以z的值也不为一定值。

y\x	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	NAC
c2	UNDEF	c3	NAC
NAC	NAC	NAC	NAC

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。（基本块的入口/出口位置变量的状态）

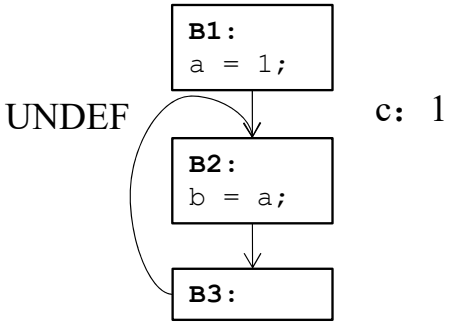
常量传播

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。

SCP的传递函数与先前所述的三种状态相同。



控制流约束函数：

控制流约束函数与传递函数略有不同：

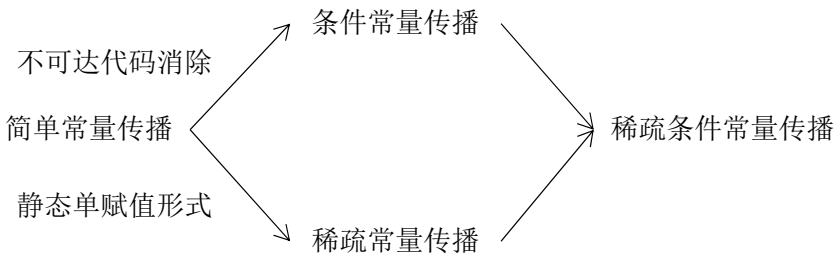
UNDEF∪c = c　　NAC∪c = NAC

c∪c = c　　c1∪c2 = 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，UNDEF∪c = c，其原因如右图

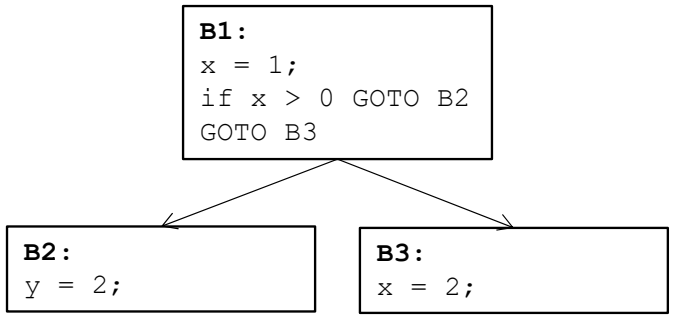
常量传播

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。



常量传播

3) 稀疏常量传播(Sparse Constant Propagation)

“稀疏”（Sparse）：以静态单赋值形式（Static Single-Assignment Form, SSA form）的中间表示形式为基础，分析变量的使用-定义关系。

使用（Use）：表达式的一个运算分量或赋值语句等号右侧的值，

定义（Define）指该变量被赋值的语句。

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

- 常量折叠：因变量仅被定义一次，暴露常量赋值，如变量f2。

- 公共子表达式消除：对于公共子表达式a1+b1，变量a1与b1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。

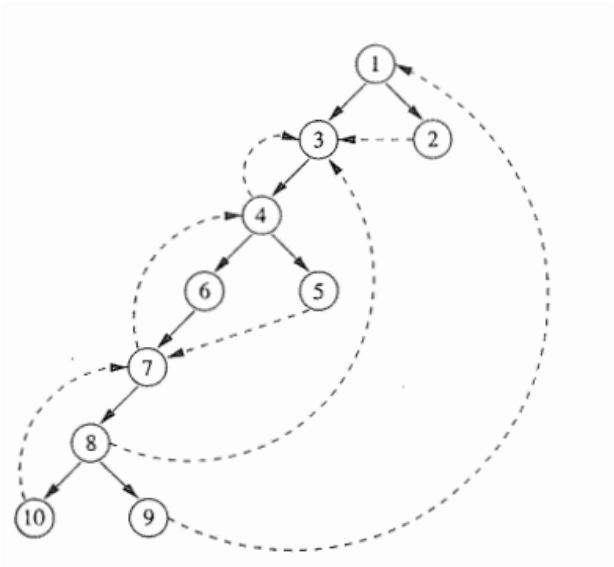
- 无用代码消除：在基本块出口不活跃的变量，若在基本块的内部仅被def而未有use，可认为其是无用代码并予以消除，如f1 := a1 + b1等。

循环不变代码外提

自然循环（Natural Loops）

1.自然循环有唯一的入口结点，称为首结点（Header），首结点支配循环内部的所有节点。

2.循环内部至少存在一条指向首结点的回边（Back Edges），若存在两节点m和n，m dom n， 存在有向边e：n→m，则有向边e被称为回边。若不存在回边，则不构成循环。



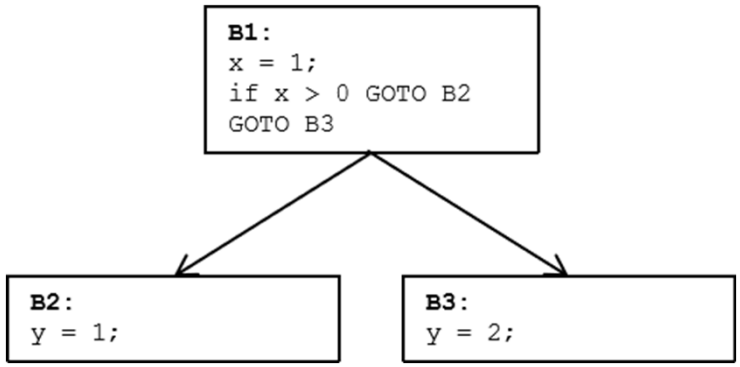
循环不变代码外提

使用的数据流分析模式：到达定值

循环中有哪些代码可以外提？循环内部的语句通常比其他语句执行的次数更多，如果把不必出现在循环内部的语句移动到外部，这些代码就可只执行一次，能起到较好的优化效果。

不能被移动到循环外侧的语句：

1. 被移动前是不可达代码



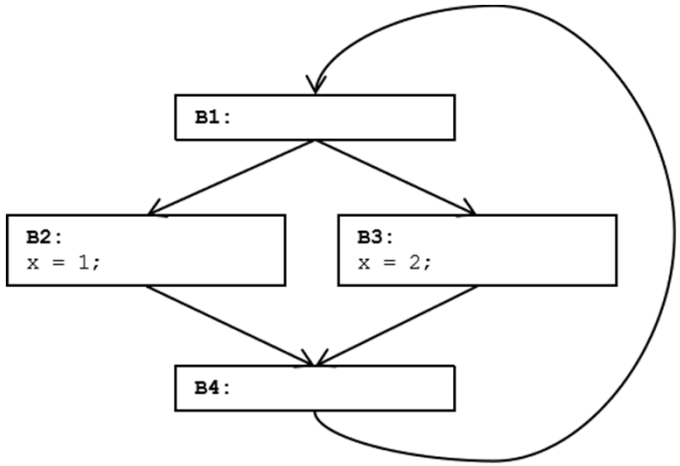
循环不变代码外提

使用的数据流分析模式：到达定值

循环中有哪些代码可以外提？循环内部的语句通常比其他语句执行的次数更多，如果把不必出现在循环内部的语句移动到外部，这些代码就可只执行一次，能起到较好的优化效果。

不能被移动到循环外侧的语句：

2. 移动前，循环内部对该变量的赋值是唯一的



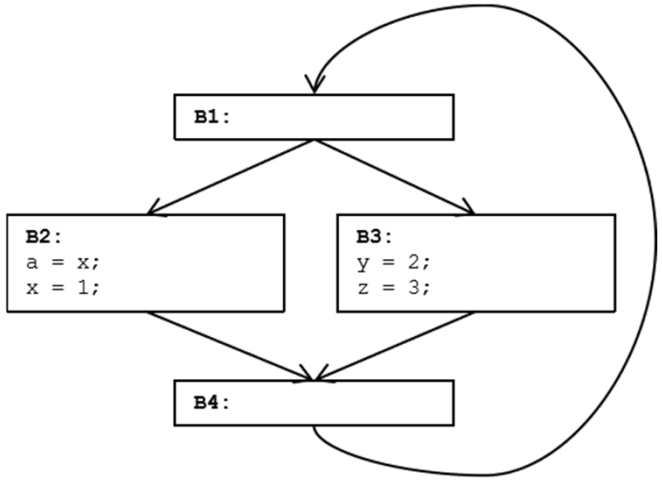
循环不变代码外提

使用的数据流分析模式：到达定值

循环中有哪些代码可以外提？循环内部的语句通常比其他语句执行的次数更多，如果把不必出现在循环内部的语句移动到外部，这些代码就可只执行一次，能起到较好的优化效果。

不能被移动到循环外侧的语句：

3. 对变量进行def之前，循环内部存在use



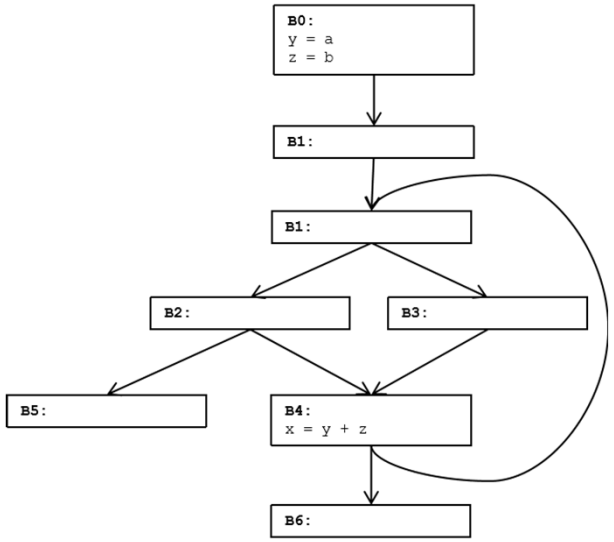
循环不变代码外提

使用的数据流分析模式：到达定值

循环中有哪些代码可以外提？循环内部的语句通常比其他语句执行的次数更多，如果把不必出现在循环内部的语句移动到外部，这些代码就可只执行一次，能起到较好的优化效果。

不能被移动到循环外侧的语句：

4. 变量定义的基本块不能支配所有的循环出口



循环不变代码外提

使用的数据流分析模式：到达定值

循环中有哪些代码可以外提？循环内部的语句通常比其他语句执行的次数更多，如果把不必出现在循环内部的语句移动到外部，这些代码就可只执行一次，能起到较好的优化效果。

可移动的代码需要满足以下的特性：

1. 循环不变
2. 所处的基本块能够支配所有的出口基本块
3. 循环内部不存在其他对该变量的赋值
4. 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

循环不变代码外提

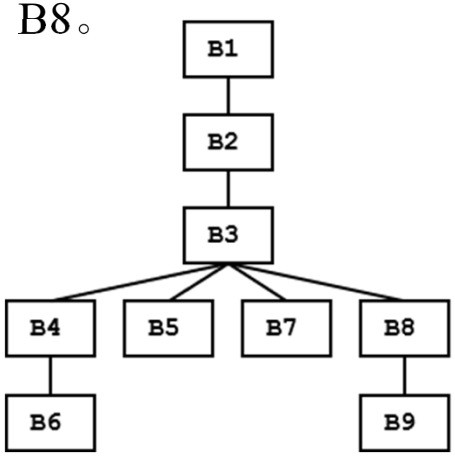
使用的数据流分析模式：到达定值

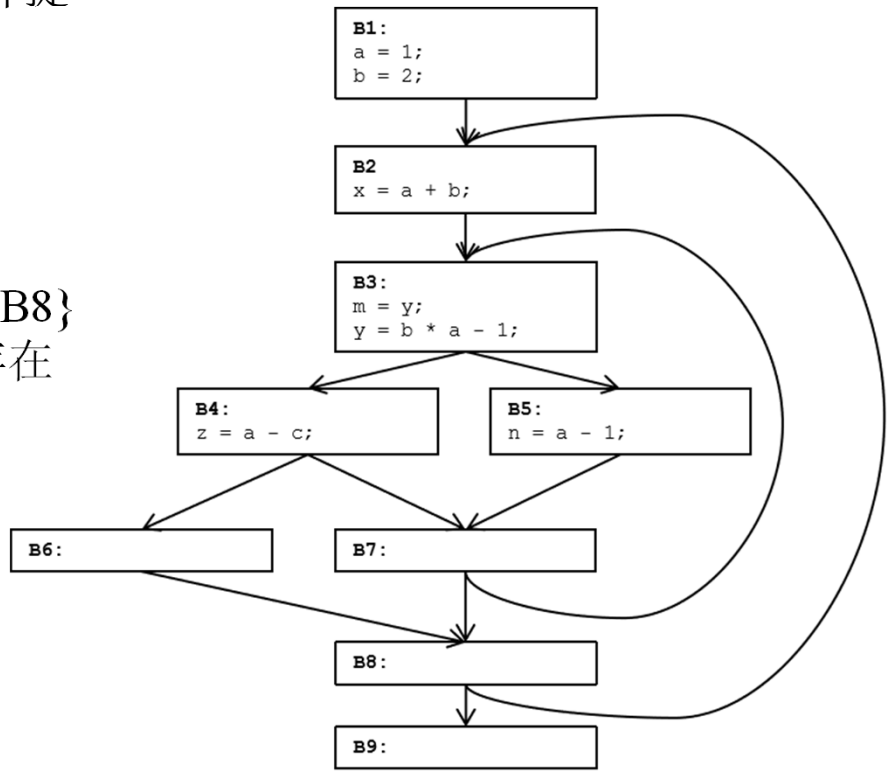
循环不变代码外提：从最内层的循环逐渐向外，进行代码外提

以右边的CFG为例，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。





循环不变代码外提

使用的数据流分析模式：到达定值

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。

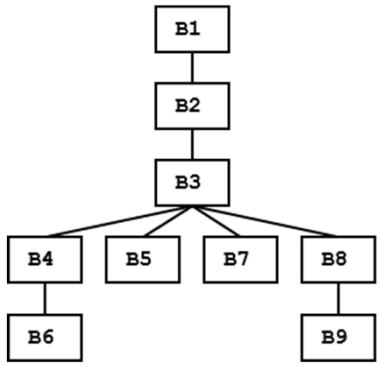
3. 变量的def关系能够支配所有对该变量的use:

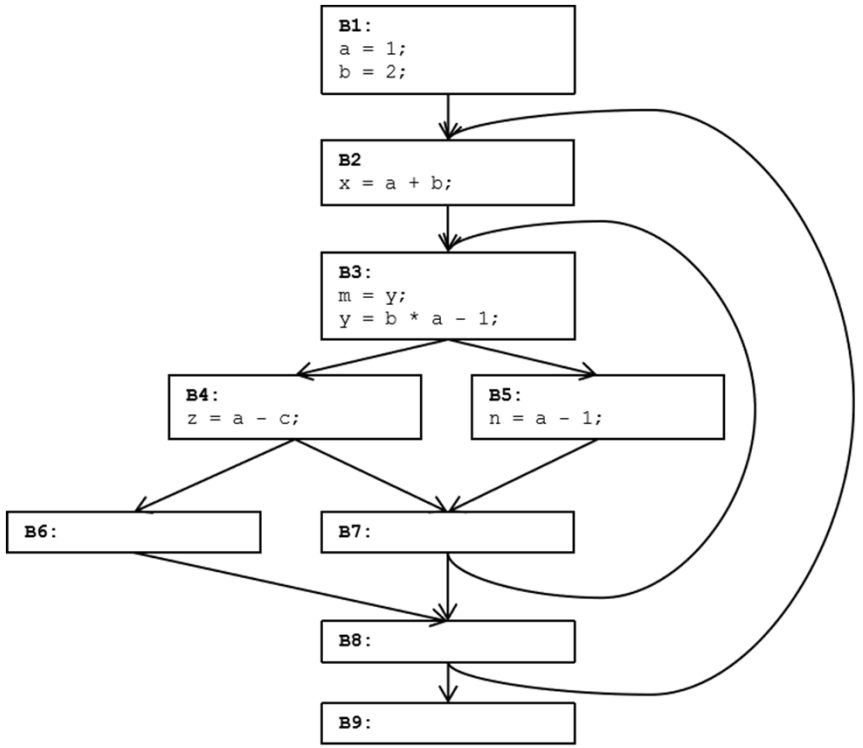
变量y的赋值不能支配所有的use。

4. 使用到达定值分析所有变量：

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。





循环不变代码外提

使用的数据流分析模式：到达定值

1. 循环L2内部能够支配所有出口的基本块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。

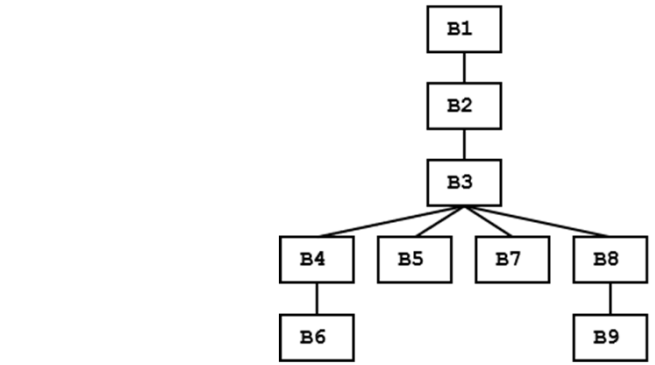
3. 变量的def关系能够支配所有对该变量的use:

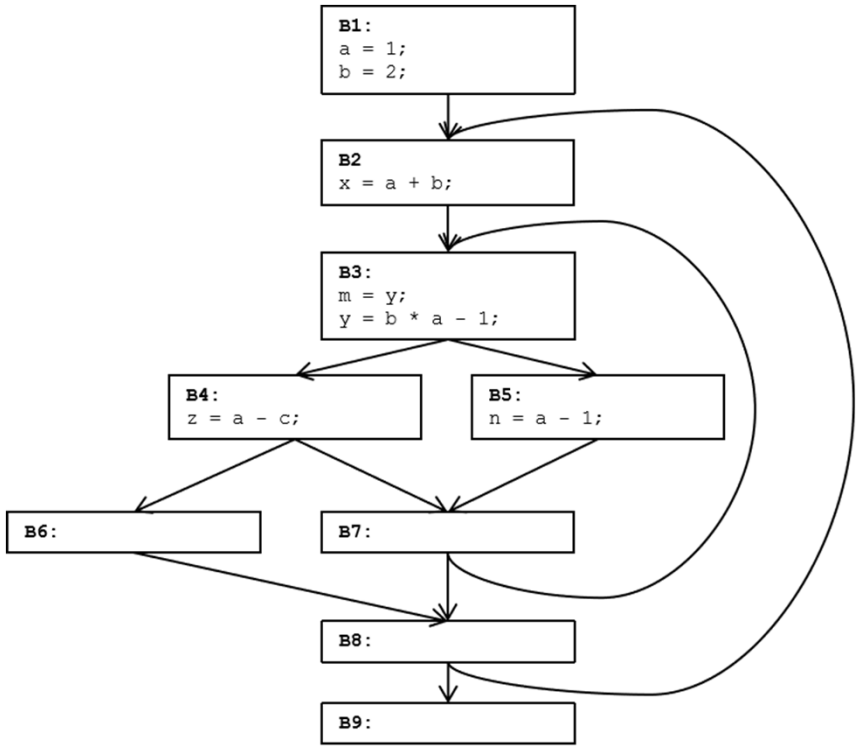
基本块B3中的变量已被分析过。

基本块B2的变量x在循环中不存在use，因此满足条件。

4. 使用到达定值分析所有变量：

赋值语句x = a + b等号右边的表达式a+b中存在的变量a与b，使用到达定值分析后，其值均为定值，因此可作为循环不变的代码外提。





归纳变量强度削减

使用的数据流分析模式：到达定值

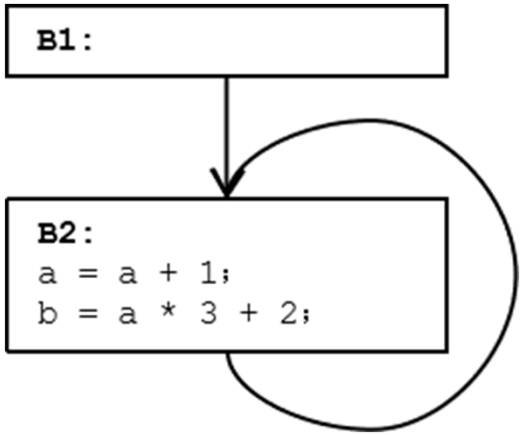
基础归纳变量（Basic Induction Variable）是在循环内部每次赋值都加或减一个常数c，并且该变量不能被替换成常数，且不是循环不变的变量，如：

x=x+c 或 x=x-c，c为一常量

归纳变量（Induction Variable）可能是：

1.一个基础归纳变量x；

2.在循环中仅有一条针对该变量的赋值语句，且赋值语句等号右边的表达式为一个基础归纳变量的线性方程组，形如y=x*c1+c2，其中c1与c2为常量。



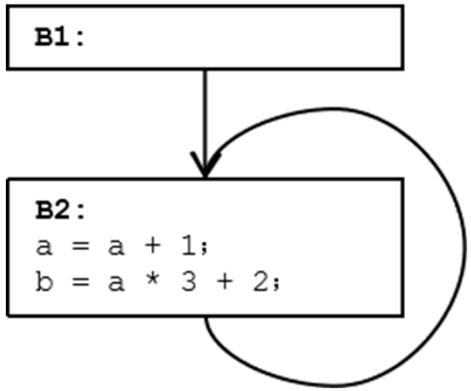
归纳变量强度削减

使用的数据流分析模式：到达定值

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 op c1 + c2，其中op表示任何二元运算

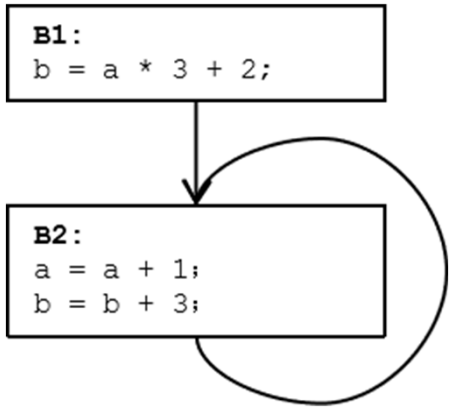
当程序内部存在右侧语句：



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：



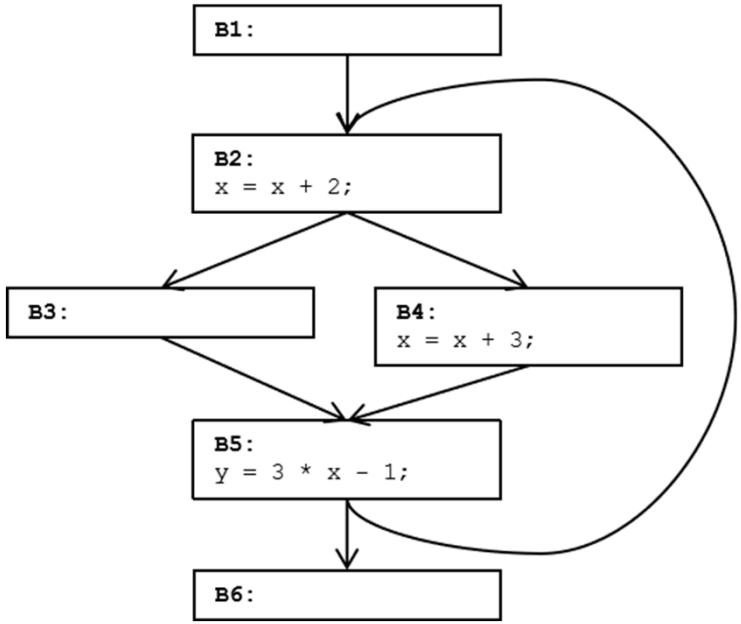
归纳变量强度削减

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。



X的赋值不能支配B5中的y，故不能进行强度削减

归纳变量强度削减

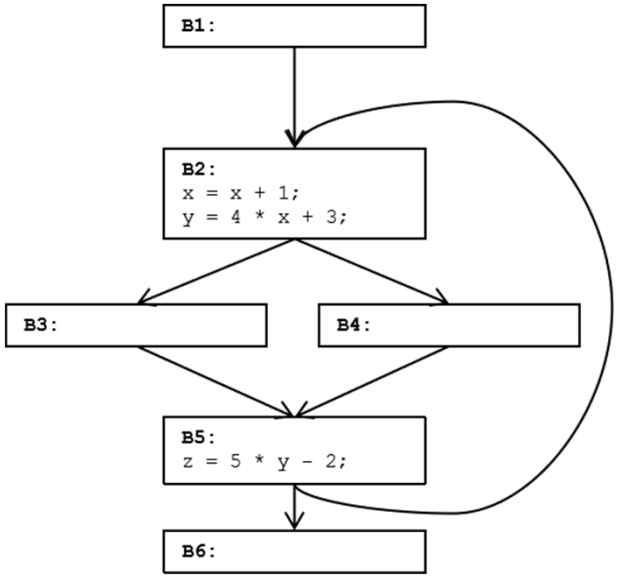
强度削减的例子：

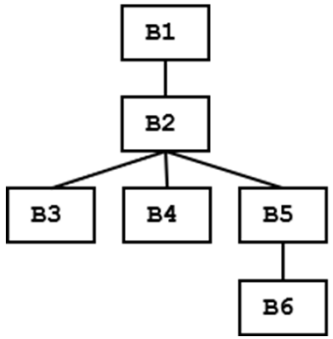
1.构造支配树；

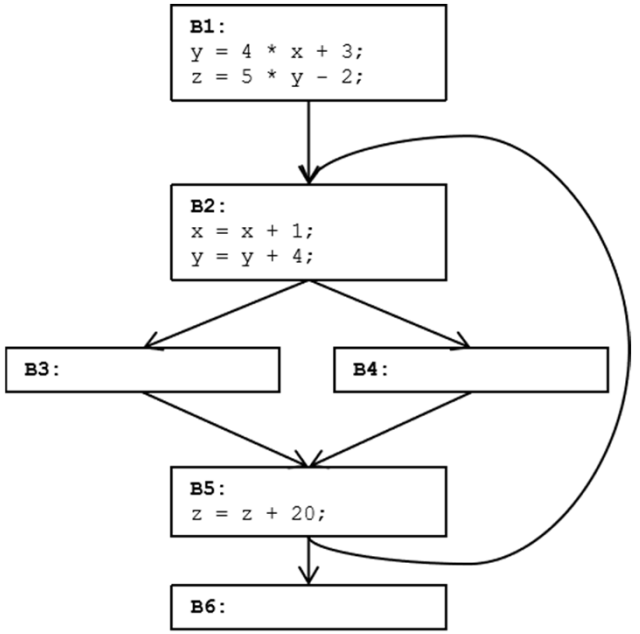
2.寻找基础归纳变量，寻找到变量x；(不能被常量传播替换成定值，每次赋值都加或减一个常数c）

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。







基于支配树（Dominator Tree）的SSA生成算法

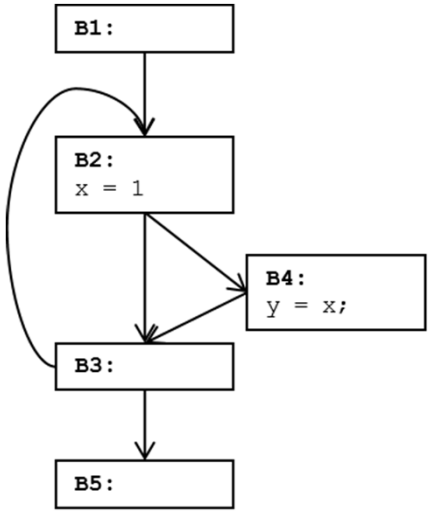
支配节点：对于某节点n，如果从入口节点到n的每一条路径都必须经过节点m，则节点m支配节点n，记为m dom n，每一个节点都支配自己。

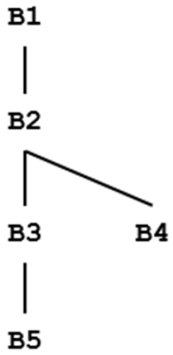
支配关系满足自反性、反对称性和传递性：

自反性：a dom a；

反对称性： a dom b, b dom a => a=b；

传递性：a dom b, b dom c => a dom c。

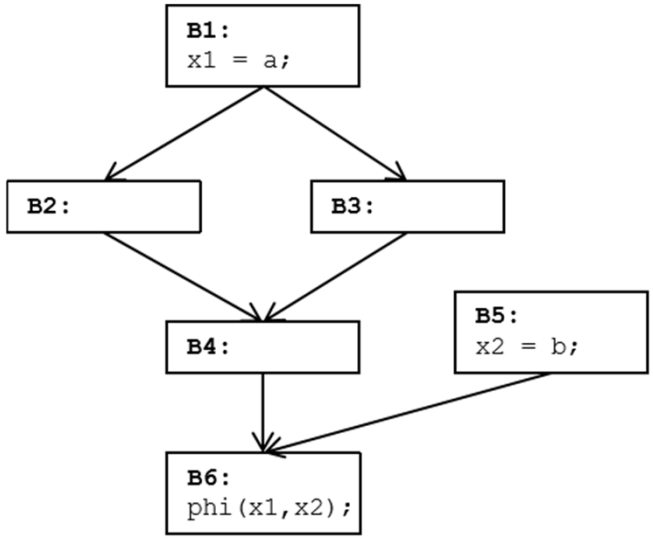




基于支配树（Dominator Tree）的SSA生成算法

支配边界（Dominance Frontier）： A支配B的一个前驱节点但不严格支配B，B是A的支配边界

在支配边界上放置一个phi函数，合并不同路径上的程序状态



如何计算支配边界？

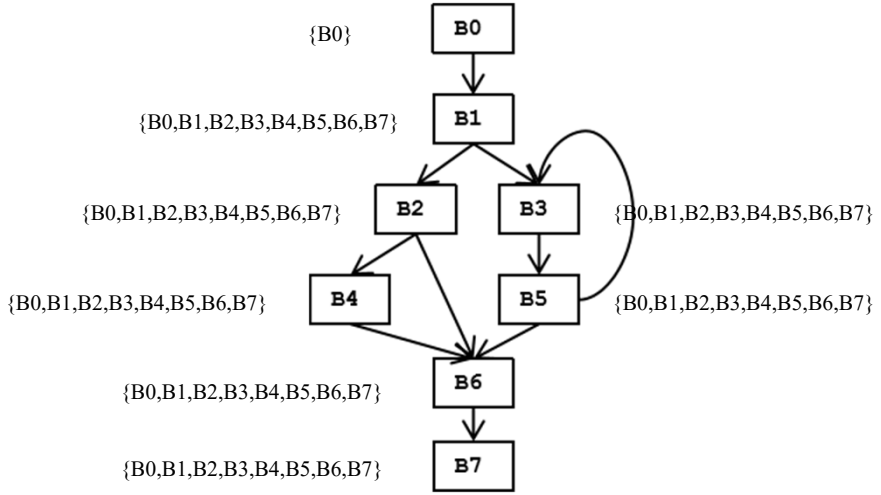
基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

1）计算支配关系。

支配情况更新函数：

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

2.是n本身 {n}。

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}

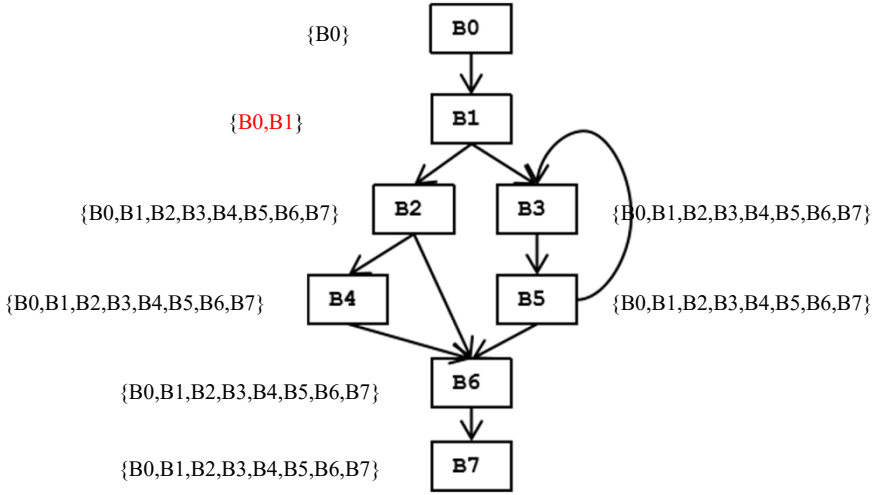
基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

1）计算支配关系。

支配情况更新函数：

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

2.是n本身 {n}。

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}

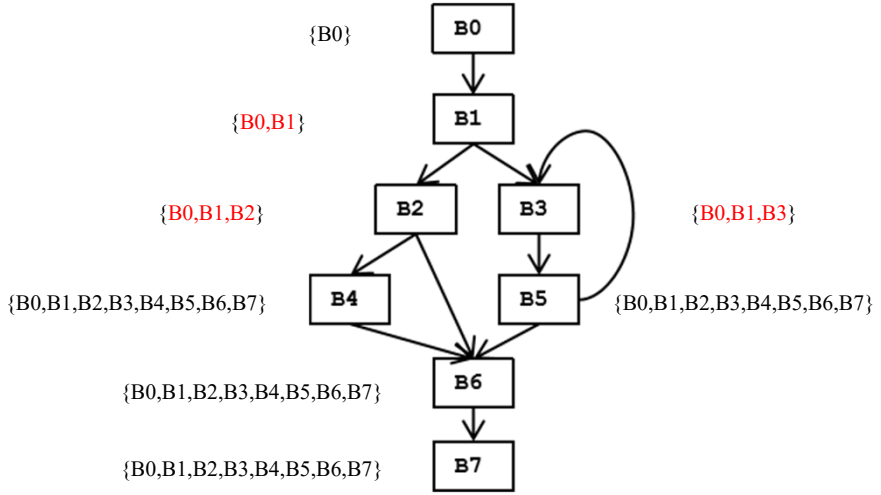
基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

1）计算支配关系。

支配情况更新函数：

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

2.是n本身 {n}。

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}

基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

1）计算支配关系。

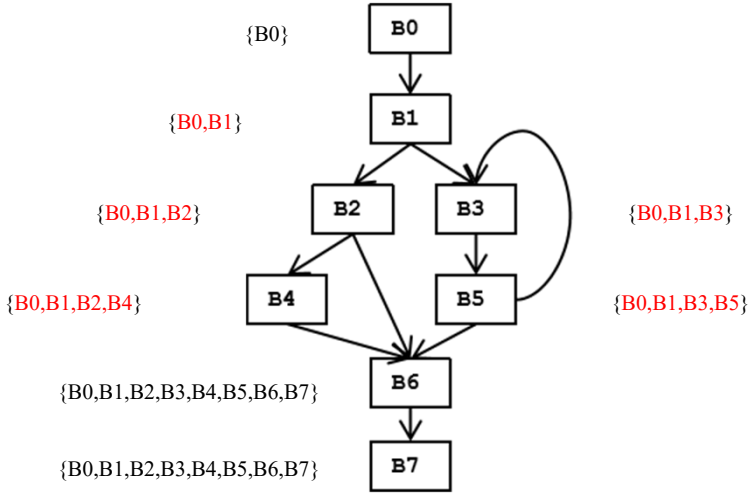
支配情况更新函数：

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

2.是n本身 {n}。



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}

基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

1）计算支配关系。

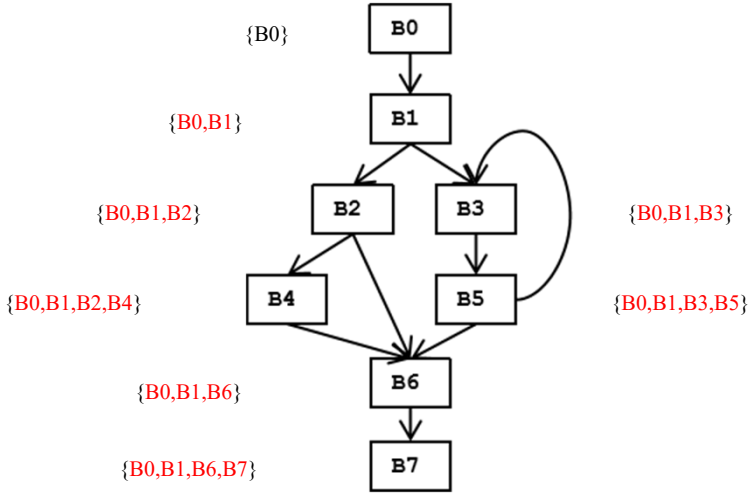
支配情况更新函数：

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

2.是n本身 {n}。



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}

基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

2）基于支配关系，构造支配树。D(n) 支配该节点的 → dom(n) 该节点支配的

3）构造严格支配表, 把当前节点剔除 （因内部的语句只支配出口，不支配入口）

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}

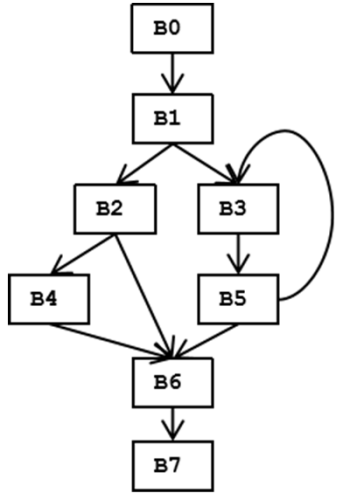
➡

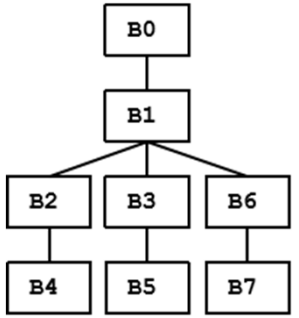
block	D(n)
B0	{B0}
B1	{B0,B1}
B2	{B0,B1,B2}
B3	{B0,B1,B3}
B4	{B0,B1,B2,B4}
B5	{B0,B1,B3,B5}
B6	{B0,B1,B6}
B7	{B0,B1,B6,B7}

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

➡

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	{}





基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

4）寻找支配边界

支配边界：基本块n的后继中，第一个不被基本块n支配的基本块

即支配的所有节点的直接后继集合中，不被n支配的节点

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	{}

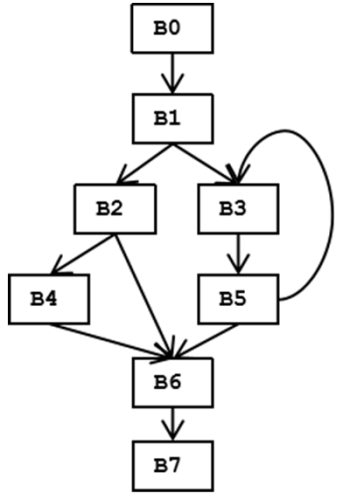
严格支配的节点

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	{}

支配的所有节点的直接后继集合

支配边界： succ(dom(n)) - sdom(n)

从支配的所有节点直接后继集合中剔除严格支配的节点



block	
B0	{}
B1	{}
B2	{B6}
B3	{B3,B6}
B4	{B6}
B5	{B3,B6}
B6	{}
B7	{}

基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

5) 添加phi函数

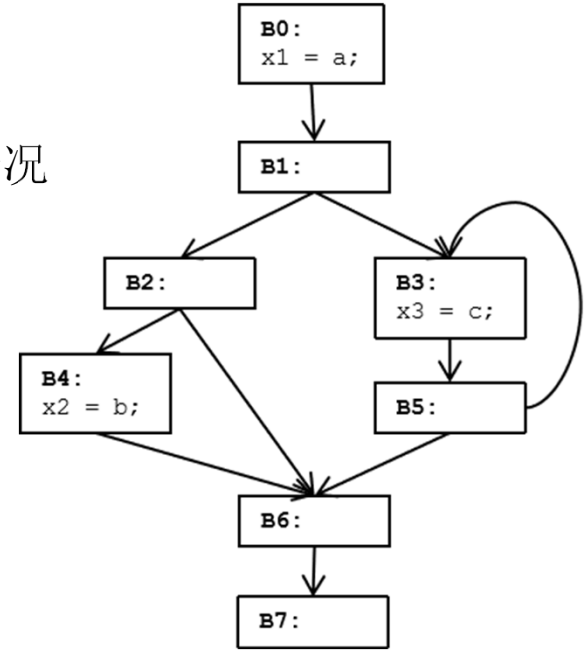
计算出支配边界之后，基于支配边界进行phi函数的添加。

变量x的赋值共有三处：B0、B3、B4

首先，我们使用到达定值，计算在每个基本块入口处，x的可能取值情况

B0	B1	B2	B3	B4	B5	B6	B7
{x1}	{x1}	{x1}	{x1, x3}	{x1}	{x1, x3}	{x1, x2, x3}	{x1, x2, x3}

然后，我们使用工作表算法，进行phi语句的植入



基于支配树（Dominator Tree）的SSA生成算法

以右侧CFG为例

5) 添加phi函数

B0	B1	B2	B3	B4	B5	B6	B7
{x1}	{x1}	{x1}	{x1, x3}	{x1}	{x1, x3}	{x1, x2, x3}	{x1, x2, x3}

然后，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 : {B0、B3、B4}

1.从w中取得基本块B0，B0的支配边界集合为{}，不添加phi语句

w : {B3、B4}

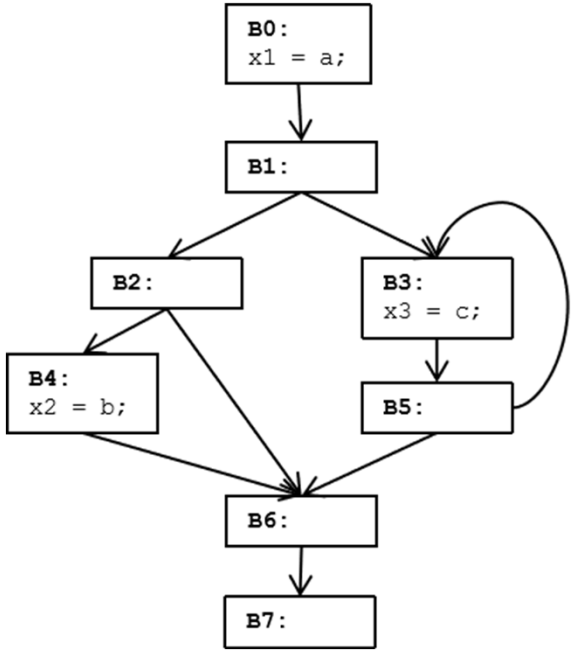
2.从w中取得基本块B3，B3的支配边界集合为{B3、B6}，在基本块B3和B6中添加phi语句。B3中添加的phi语句合并x1和x3，B6中添加的phi语句合并x1、x2和x3，然后将B3和B6加入工作表中（因基本块B3已添加phi语句，因此只将B6加入到工作表w中）。

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。

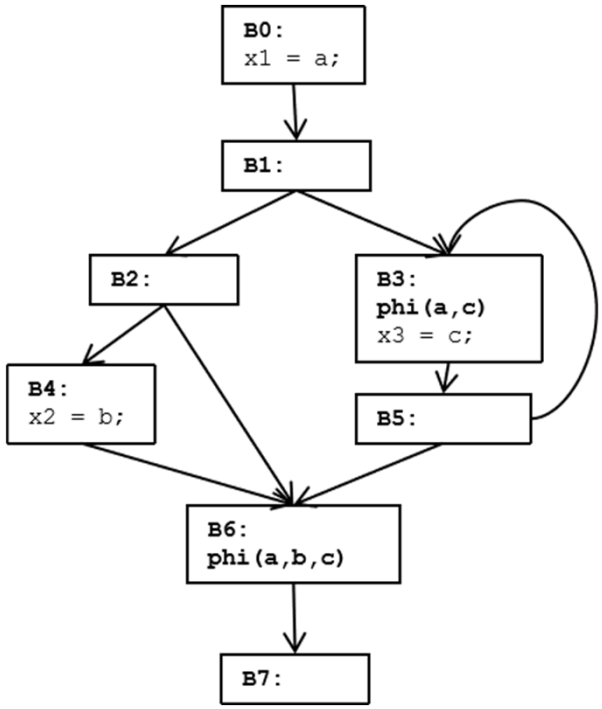


基于支配树（Dominator Tree）的SSA生成算法

以下面的CFG为例

添加phi函数后，x是否是定值一目了然

B0	B1	B2	B3	B4	B5	B6	B7
{x1}	{x1}	{x1}	{x1, x3}	{x1}	{x1, x3}	{x1, x2, x3}	{x1, x2, x3}



中间代码优化

2025

南京大学计算机学院