



软件分析

数据流分析：基础

熊英飞
北京大学



示例：符号分析

- 给定一个只包含整数变量和常量的程序，已知输入的符号，求输出的符号



回顾： 表达式上的抽象和近似

输入 x 和 y 都是正数
求表达式 $x+1-1+y$ 的符号

抽象符号

正 = {所有的正数}

零 = {0}

负 = {所有的负数}

𐀀 = {所有的整数和NaN}

表达式	具体值范围	抽象值	抽象值范围
x	$[1, +\infty]$	正	$[1, +\infty]$
$x+1$	$[2, +\infty]$	正	$[1, +\infty]$
$x+1-1$	$[1, +\infty]$	𐀀	$[-\infty, +\infty]$
y	$[1, +\infty]$	正	$[1, +\infty]$
$x+1-1+y$	$[2, +\infty]$	𐀀	$[-\infty, +\infty]$

输入是集合时，表达式返回值也是集合。精确集合难以直接计算，
所以用抽象值来近似。抽象值需要保证覆盖精确集合的范围。



抽象域的运算和正确性

- 引入 γ 表示抽象值对应的集合

- $\gamma(\text{正}) = \{x \in \text{整数} \mid x > 0\}$

- α 将具体域的值映射到抽象域

- $\alpha(1) = \text{正}$

- 需要满足: $x \in \gamma(\alpha(x))$

- 运算操作 $+$, $-$, \times , $/$ 分别定义抽象域上的版本 \oplus , \ominus , \otimes , \oslash

- 需要满足输入抽象域包含的任何具体值计算结果在输出抽象域中

- 如: $\forall x \in \gamma(\text{甲}) \wedge y \in \gamma(\text{乙}), x + y \in \gamma(\text{甲} \oplus \text{乙})$

\oplus	正	负	零	罅
正	正			
负	罅	负		
零	正	负	零	
罅	罅	罅	罅	罅



程序 vs 表达式

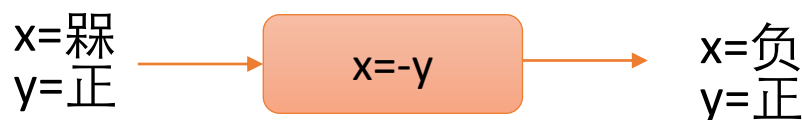
- 程序中多了如下概念，如何用抽象方法进行分析？
 - 语句
 - 顺序
 - 分支
 - 循环
- 处理这一系列程序中抽象概念的方法叫做数据流分析
 - 数据流分析将程序转换成控制流图，并在控制流图上完成分析

暂时不考虑指针、数组、结构体、函数调用、动态内存分配等高级编程语言成分，将在未来课程中处理



从表达式到语句

- 抽象域表示系统状态中值的集合
 - 系统状态：从变量名到值的映射
 - 抽象状态：从变量名到抽象值的映射
- 例：如果 y 为正数，执行 $x=-y$ 之后， x 的符号是什么？



表示 x 为任意值， y 为正数的
所有系统状态的集合

- $x=-1, y=1$
- $x=0, y=1$
- $x=1, y=1$
- $x=1, y=2$
- $x=100, y=200$
-

表示 x 为负数， y 为正数的所有
系统状态的集合

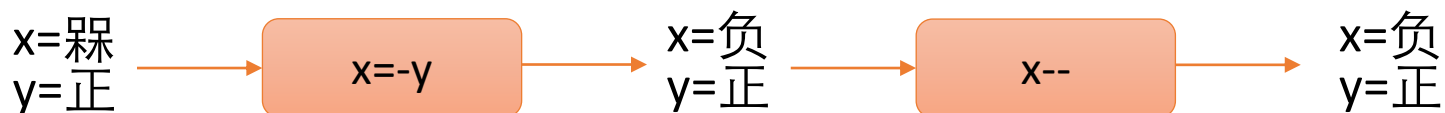
- $x=-1, y=1$
- $x=-2, y=2$
- $x=-200, y=200$
- $x=-1, y=2$
- $x=-2, y=1$
-

正确性：输入抽象值包含的任意具体状态的执行结果在输出抽象值中



语句的顺序组合

- 依次执行语句即可
- 例：如果 y 为正数，执行 $x=-y; x--;$ 之后， x 的符号是什么？



正确性：输入抽象值包含的任意具体状态的执行结果在输出抽象值中



分支

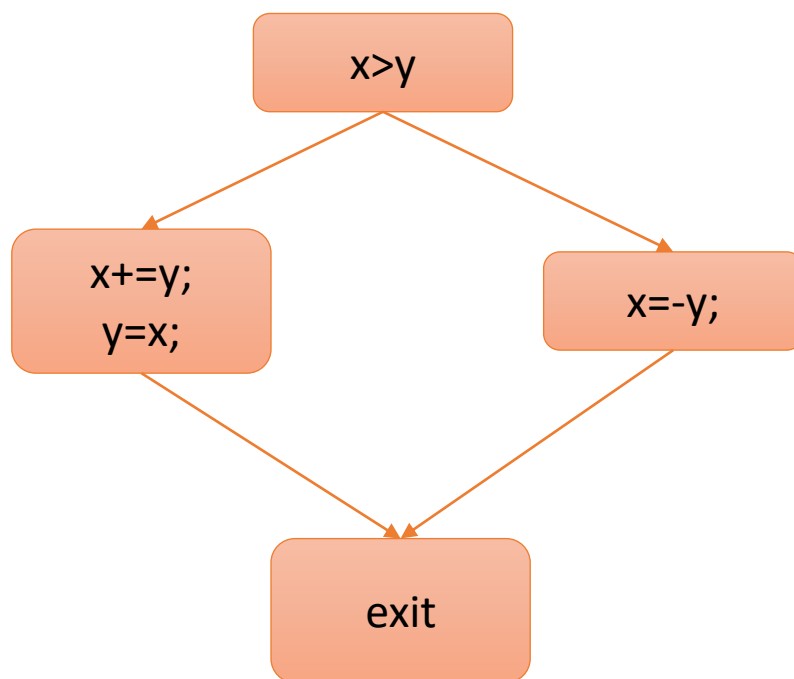
- 例

```
if (x>y) {  
    x+=y;  
    y = x; }  
else {  
    x = -y;  
}
```

- 输入:

- x=正
- y=正

- 求输出的符号



输入抽象值对应具体状态左右路径都有可能走



分支

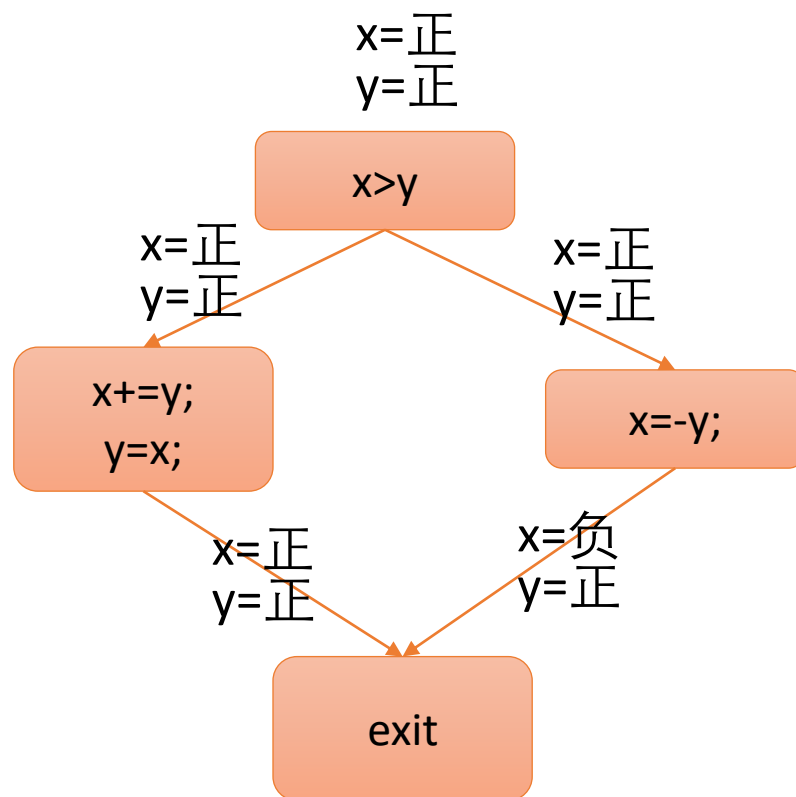
- 例

```
if (x>y) {  
    x+=y;  
    y = x;  
}  
else {  
    x = -y;  
}
```

- 输入:

- $x = \text{正}$
- $y = \text{正}$

- 求输出的符号



针对两条路径分别计算



分支

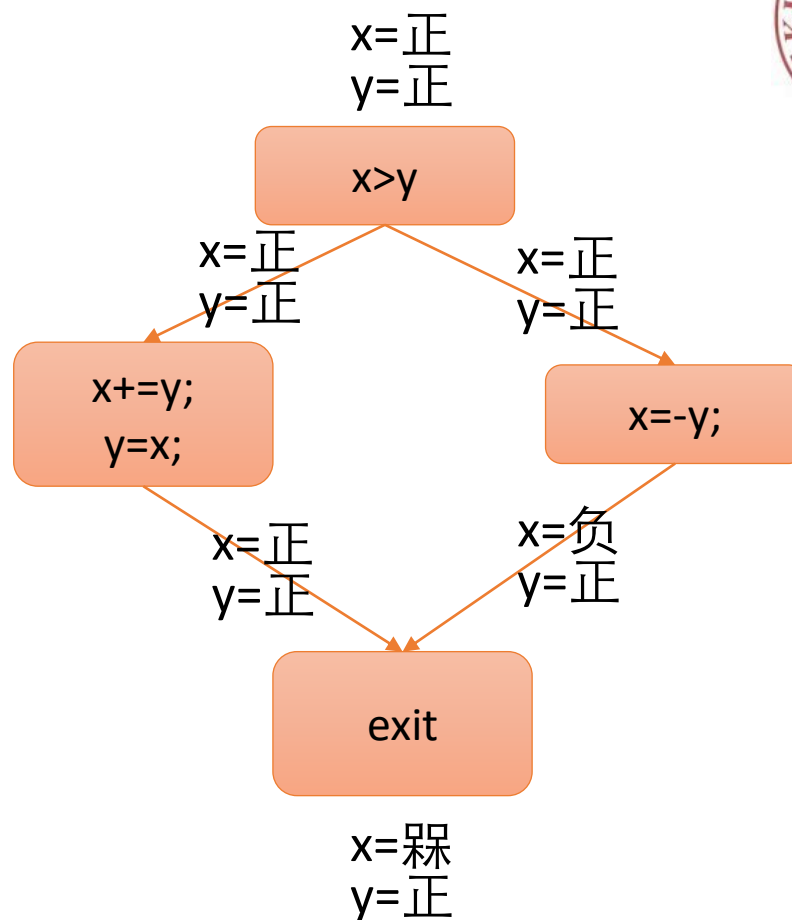
- 例

```
if (x>y) {  
    x+=y;  
    y = x;  
} else {  
    x = -y;  
}
```

- 输入:

- $x = \text{正}$
- $y = \text{正}$

- 求输出的符号



定义合并操作:

$$\sqcup (v_1, v_2) = \begin{cases} v_1 & \text{如果 } v_1 = v_2 \\ \text{赧} & \text{其他情况} \end{cases}$$

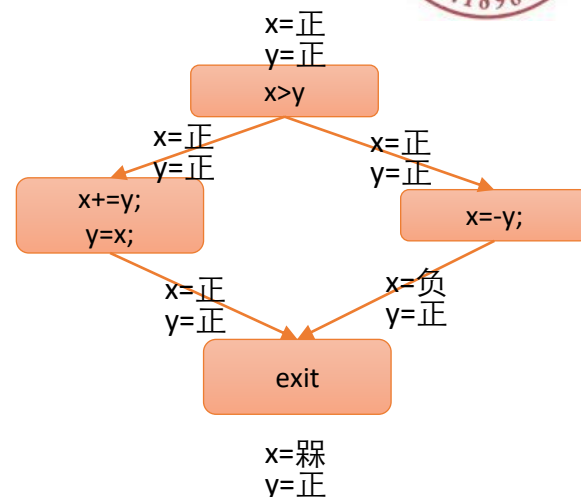


分支分析正确性

- 输入抽象值包含的任意具体状态的执行结束状态在输出抽象值中

• 证明:

1. 对任意具体状态，只会走单一路径
2. 分叉时，抽象值不变，所以该具体状态一定被包含在对应分支的路径上
3. 任一支一定保证执行结束状态在输出抽象值中
4. 合并时， \sqcup 函数保证输入对应的具体状态一定包含在结果中
即 $\forall x \in \gamma(\text{甲}), x \in \gamma(\text{甲} \sqcup \text{乙})$ ，且 \sqcup 满足交换律





循环

```
x*=-100;
```

```
y+=1;
```

```
while(y < z) {
```

```
    x *= -100;
```

```
    y += 1;
```

```
}
```

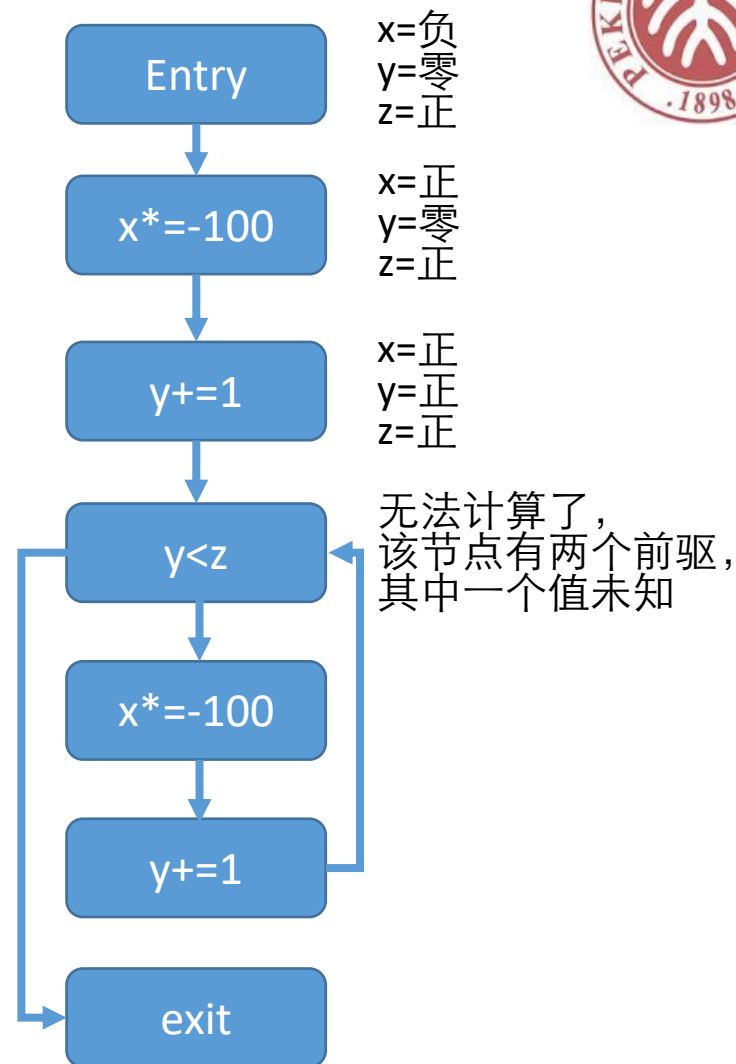
输入：x为负，y为零，z为正

输出：x为负，y为正，z为正



循环——思路

- 循环在控制流图上也是表现为分叉和合并两种
- 采用分支同样的方式处理





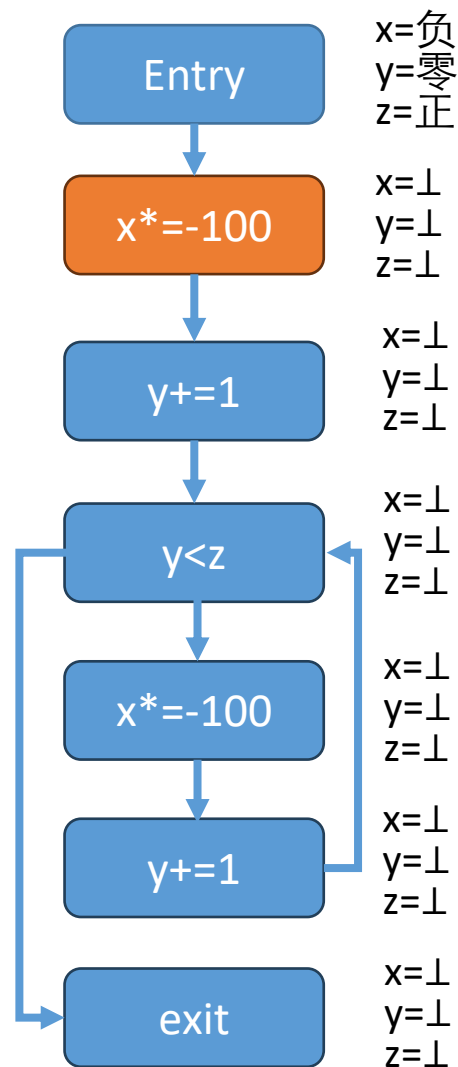
循环——解决方案

- 引入抽象值 \perp
 - $\gamma(\perp) = \emptyset$
- \sqcup 操作扩展到 \perp : $x \sqcup \perp = x$
- 初始节点上的抽象值都为 \perp ，表示没有对应具体状态
- 在分析过程中逐步扩大抽象值，加入可能的具体状态
- 如果抽象值不再增大，则分析结束



循环

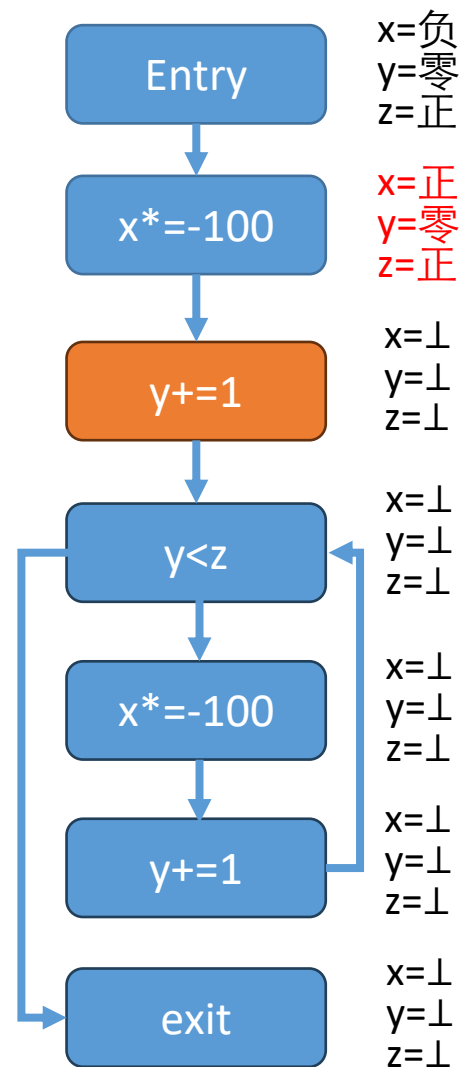
- 一开始
 - 输入节点为输入的抽象值
 - 其他节点的抽象值都为空
- 把输入节点的后继都加入待更新集合





循环

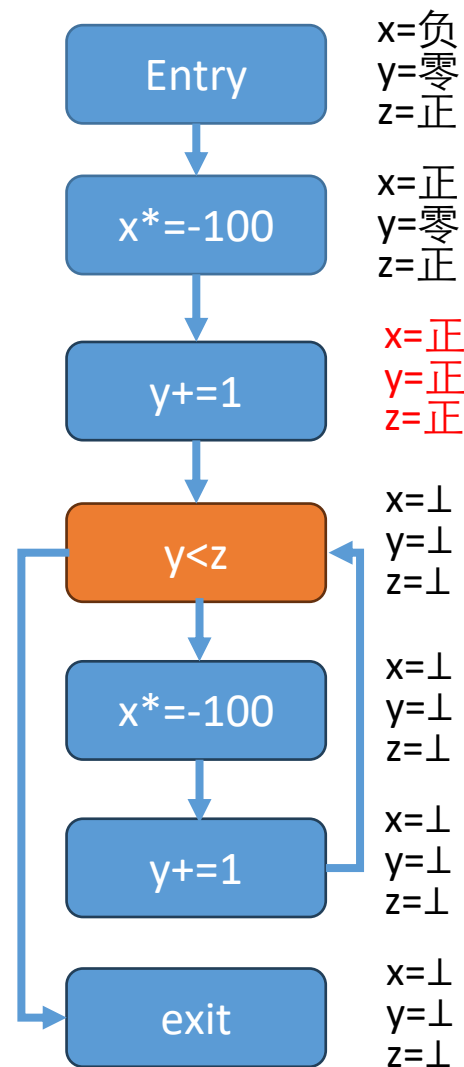
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合





循环

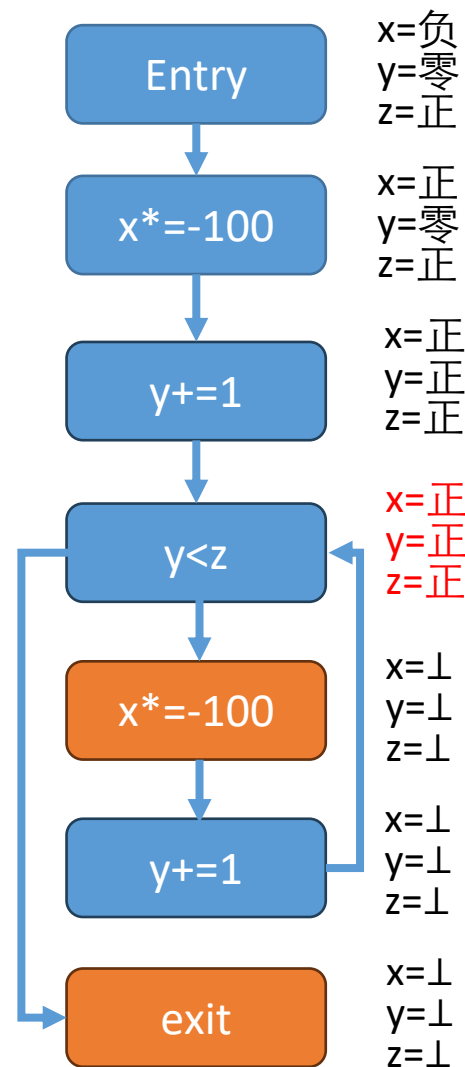
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合





循环

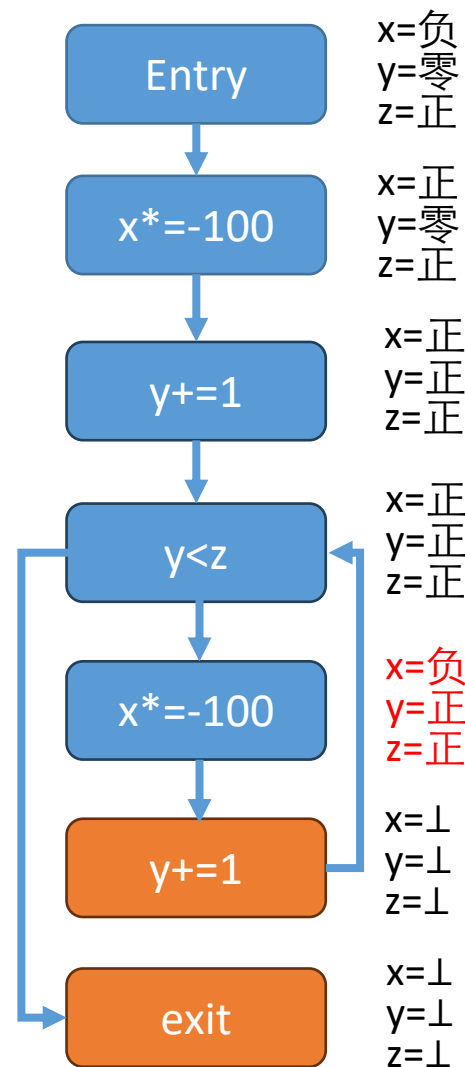
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合





循环

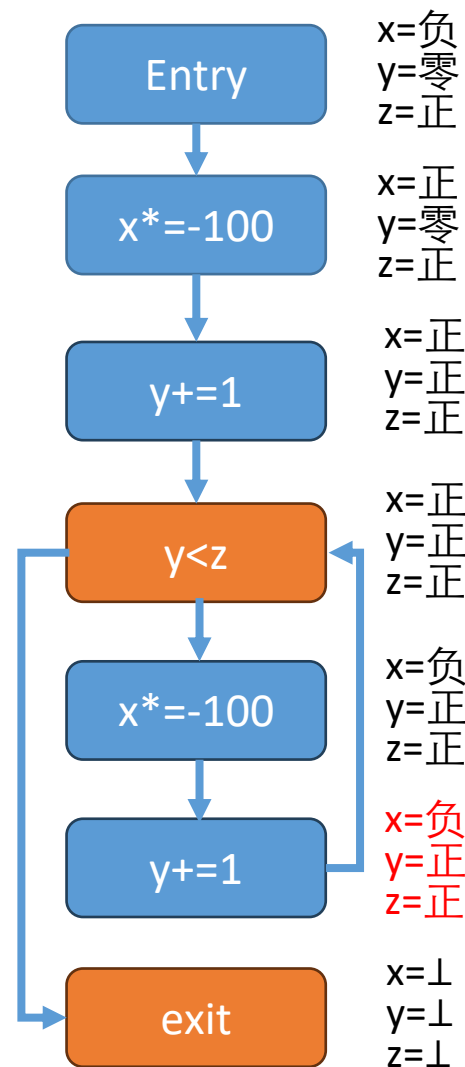
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合





循环

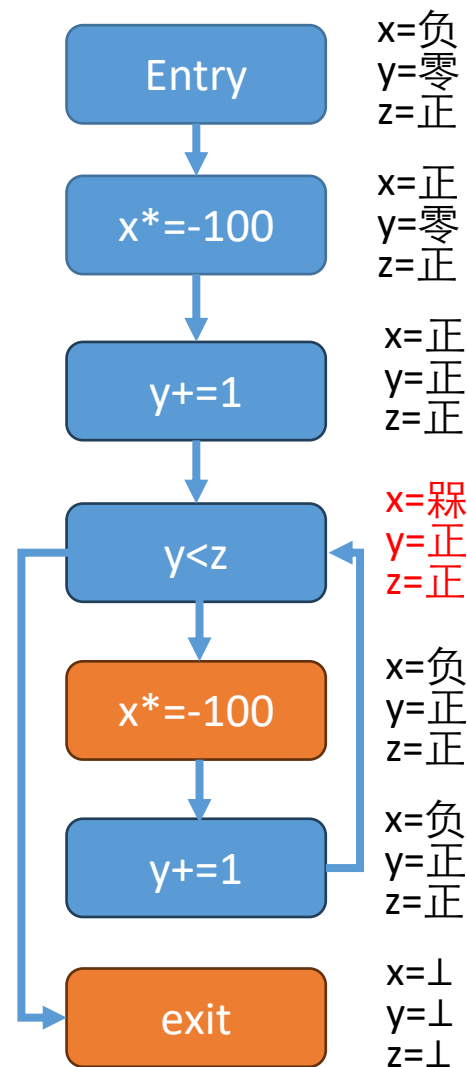
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合
- 反复迭代直到没有节点需要更新





循环

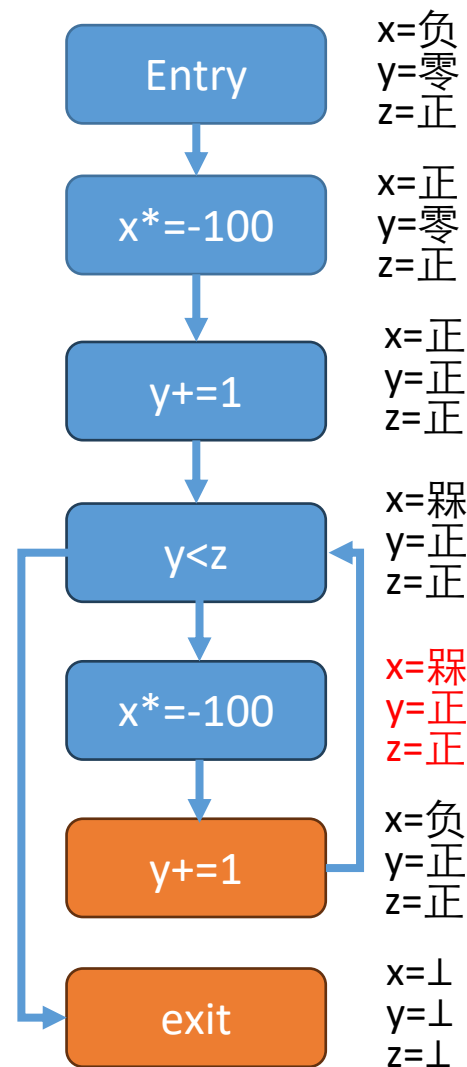
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合
- 反复迭代直到没有节点需要更新





循环

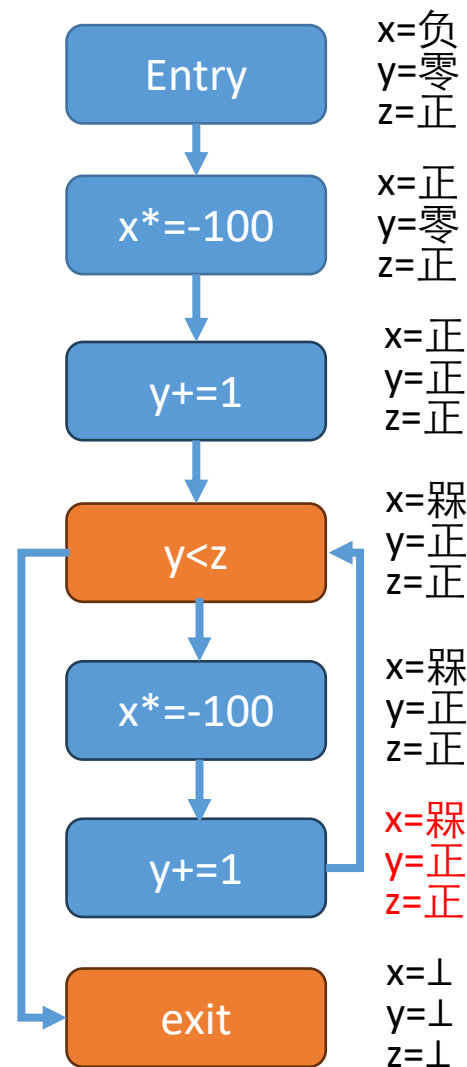
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合
- 反复迭代直到没有节点需要更新





循环

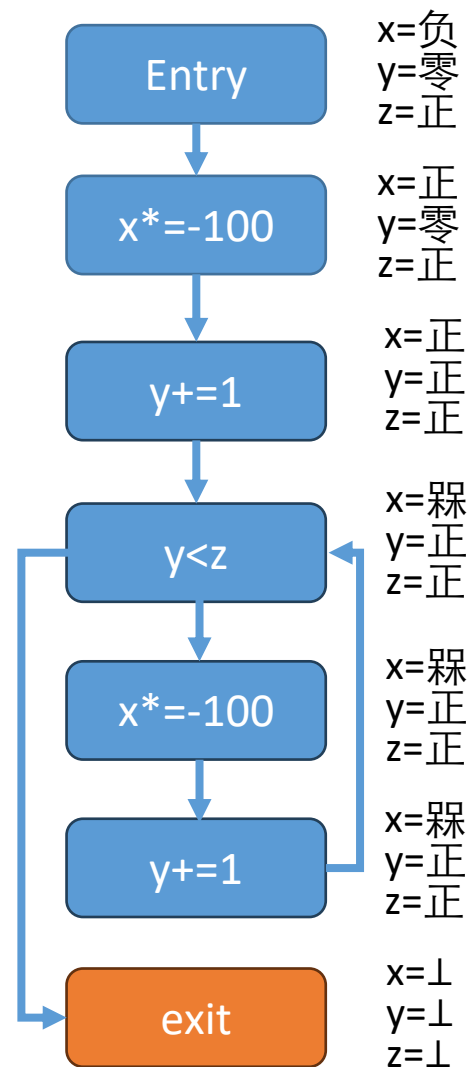
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合
- 反复迭代直到没有节点需要更新





循环

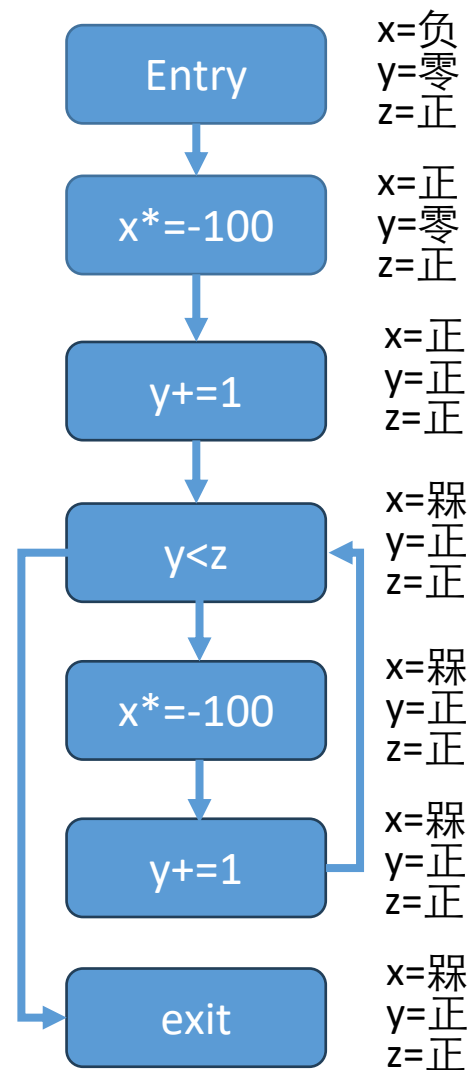
- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合
- 反复迭代直到没有节点需要更新





循环

- 每次选择一个节点更新
- 如果值有变化，该节点的后继节点也需要加入待更新集合
- 反复迭代直到没有节点需要更新





符号分析-算法

- 令 $\mathbf{S} = \{(s_x, s_y, s_z) \mid s_x, s_y, s_z \in \{\text{正, 负, 零, 罅, } \perp\}\}$
- 每个节点的值为 \mathbf{S} 的一个元素，代表对应语句执行之后的抽象状态，用 OUT 表示
 - 同时用 IN 表示语句执行之前的抽象状态
- 初始值
 - $\text{OUT}_{\text{entry}} = (\text{负}, \text{零}, \text{正})$
 - $\text{OUT}_{\text{其他节点}} = (\text{T}, \text{T}, \text{T})$
- 节点转换函数 $f_v: \mathbf{S} \rightarrow \mathbf{S}$
 - $f_{\text{exit}} = \text{id}$
 - $f_{\text{其他节点}}$ = 根据相应语句进行计算
- 合并运算 $\text{IN}_v = \sqcup_{w \in \text{pred}(v)} \text{DATA}_w$
- 节点更新运算 $\text{OUT}_v = f_v(\text{IN}_v)$
- 如果某个节点的前驱节点发生了变化，则使用节点更新运算更新该节点的附加值
- 如果没有任何节点的值发生变化，则程序终止。



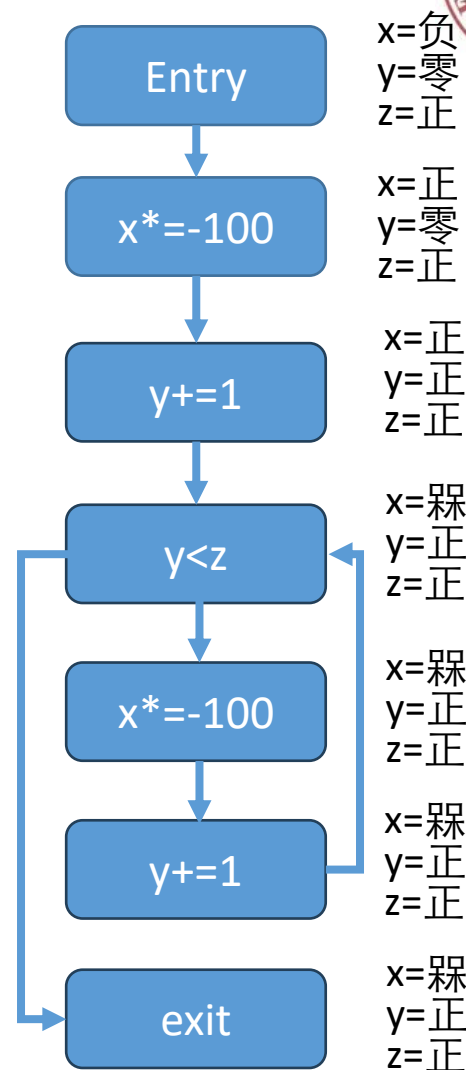
符号分析实现算法

```
OUTentry = I
 $\forall v \in (V - \text{entry}): \text{OUT}_v \leftarrow \top$ 
ToVisit  $\leftarrow \text{succ}(\text{entry})$ 
While (ToVisit.size > 0) {
    v  $\leftarrow$  ToVisit中任意节点
    ToVisit -= v
     $\text{IN}_v \leftarrow \sqcup_{w \in \text{pred}(v)} \text{OUT}_w$ 
    If ( $\text{OUT}_v \neq f_v(\text{IN}_v)$ ) ToVisit  $\cup = \text{succ}(v)$ 
     $\text{OUT}_v \leftarrow f_v(\text{IN}_v)$ 
}
```



分析为什么是对的?

- 转换函数和合并函数可以看做是定义了一组方程
 - $OUT_{entry} = (\text{负}, \text{零}, \text{正})$
 - $IN_v = \sqcup_{w \in pred(v)} DATA_w$
 - $OUT_v = f_v(IN_v)$
 - 其中 v 为 $entry$ 之外的任意节点
- 计算到不动点, 意味着上述等式全部成立, 即结果是方程的一组解





符号分析正确性

- 如果分析正常结束，给定输入抽象值对应的任意具体输入，程序执行到任意节点的状态一定包含在该节点的抽象值中。
- 证明和分支的情况类似：
 - 该具体输入的执行产生一条路径。在路径长度上做归纳，证明路径结束位置的抽象值包括具体值。
 - 路径长度为1的时候，显然成立。
- 假设路径长度为 k 的时候成立，分三种情况讨论 $k+1$ 的情形

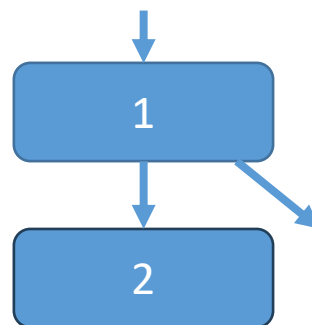
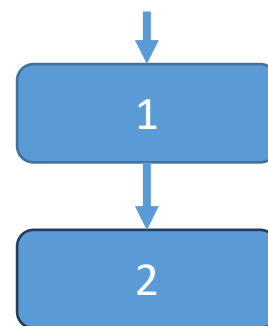
Entry

x=负
y=零
z=正



符号分析正确性

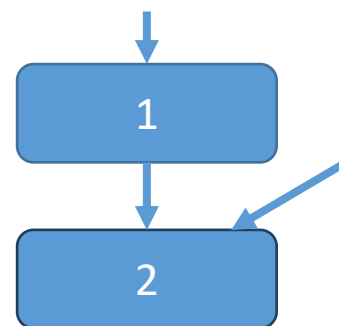
- 情况1：路径末尾只有一个后继节点，没有分叉和合并
 - $OUT_2 = f_2(OUT_1)$
 - 转换函数正确时，该情形保证正确
- 情况2：路径末尾有分叉的情况
 - $OUT_2 = f_2(OUT_1)$
 - 同上，仍然正确





符号分析正确性

- 情况3：路径末尾有合并的情况
 - $OUT_2 = f_2(OUT_1 \sqcup \text{其他值})$
 - \sqcup 保证保留所有具体状态，所以具体状态仍然在 f_2 的输入中





数据流分析的收敛性

- 该算法保证终止(Terminating)吗?
 - 路径上有环的时候，是否会一直循环？
- 该算法一定合流(Confluent)吗？
 - 有多个节点可更新的时候，是否无论先更新哪个节点最后都会到达同样的结果？
- 终止+合流=收敛(Convergence)
- 数据流分析是保证收敛的，具体证明将在数据流分析框架部分介绍



数据流分析一小结

- 近似方案1：抽象状态代表程序的多个具体状态
- 近似方案2：针对控制流节点编写转换函数
- 近似方案3：在控制流路径分叉时，复制抽象状态到所有分支
- 近似方案4：在控制流路径合并时，用 \sqcup 操作合并多个抽象状态
- 四种近似方案也代表着数据流分析的四种不精确性来源



近似方案1的不精确

- 近似方案1：抽象状态代表程序的多个具体状态
- 发生在需要将具体值集合映射到抽象值时
- 输入： x 为正数或者负数
- 抽象值： $x = \text{null}$



近似方案2的不精确

- 近似方案2：针对控制流节点编写转换函数
- 转换函数无法精确给出结果时
- 语句： $x++$
- 输入： $x = \text{正}$
- 输出： $x = \text{正}$
 - 额外引入了 $x = 1$ 的不精确情况



近似方案3的不精确

- 近似方案3：在控制流路径分叉时，复制抽象状态到所有分支
- 每个具体状态只能到达一个分支，形成不精确

$x = \text{负}$

if ($x > 0$) c1 else c2

到达c1的所有 x 都是正数

到达c2的所有 x 都是负数

$x = \text{正}$

if ($x > 0$) c1 else c2

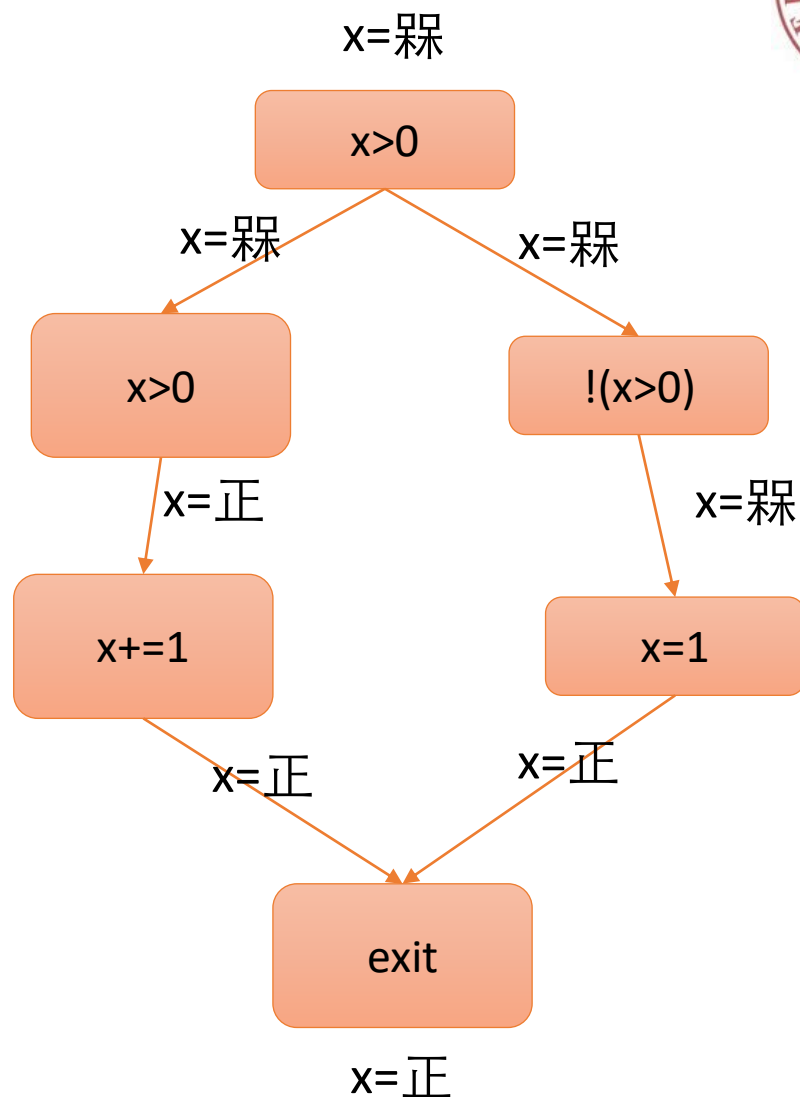
c1的抽象状态仍然是精确的

但没有状态能达到c2



解决方案

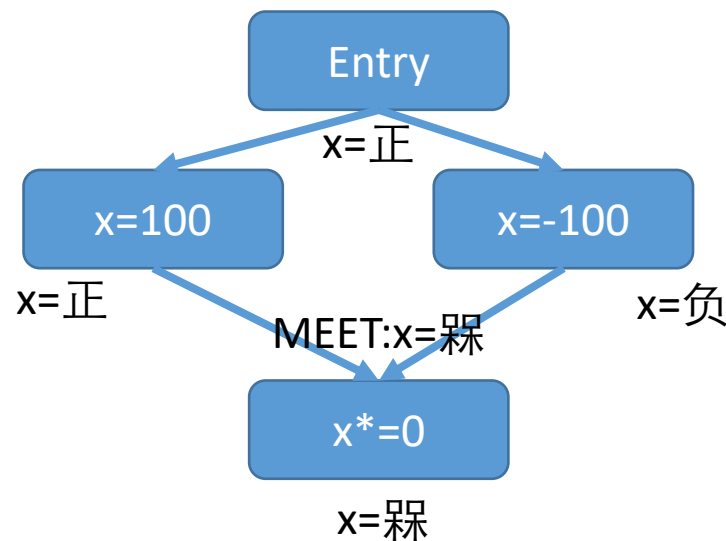
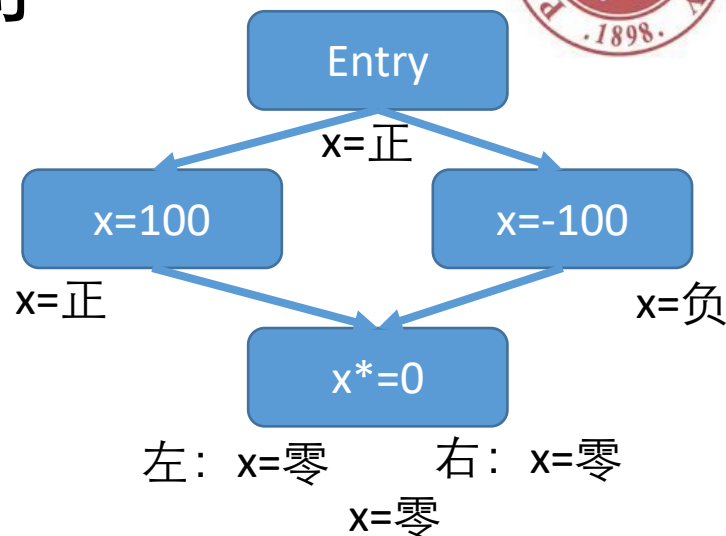
- 在每个分支添加节点，根据条件压缩抽象值
- 通常需要针对抽象域专门设计，缺少通用的方法
- 很多时候也难以压缩
 - 如右图else分支
 - 如 $x > y$ 的情况
- 精度要求不高的场合通常直接忽略分支条件





近似方案4的不精确

- 近似方案4：在控制流路径合并时，用 \sqcup 操作合并多个抽象状态
- 合并的状态可能会引入不精确值
- 右下：引入NaN、零等不精确值
- 右上：如果先计算 $*0$ ，再合并，可以得到精确答案





设计数据流分析

- 近似方案1：抽象状态代表程序的多个具体状态
 - 设计抽象域，对应的 α 、 γ 函数和初始值
- 近似方案2：针对控制流节点编写转换函数
 - 设计从基本语句导出转换函数的方法
- 近似方案3：在控制流路径分叉时，复制抽象状态到所有分支
 - 设计从条件导出压缩函数的方法（可选）
- 近似方案4：在控制流路径合并时，用 \sqcup 操作合并多个抽象状态
 - 设计 \sqcup 操作



可达定值分析

- 对程序中任意语句，分析运行该语句后每个变量的值可能是由哪些语句赋值的，给出语句标号。要求上近似，即返回值包括所有可能的定值。
 - 例：

1. <code>a=100;</code>	运行到1的时候a的定值是1
2. <code>if (c > 0)</code>	运行到2的时候a的定值是1
3. <code> a = 200;</code>	运行到3的时候a的定值是3
4. <code> b = a;</code>	运行到4的时候a的定值是3， b的定值是4
5. <code> return a;</code>	运行到5的时候a的定值是1， 3， b的定值是4



可达定值分析-抽象域

- 抽象域：从变量到指针集合的映射
 - $a \rightarrow \{1\}, b \rightarrow \{\}$
 - $a \rightarrow \{1, 3\}, b \rightarrow \{4\}$
- 初始值：所有变量都映射到空集
- 抽象值含义：具体执行序列集合，定值语句落在对应范围内
 - 具体执行序列：(语句编号, 状态)构成的序列
 - $\gamma(a \rightarrow \{1, 3\}, b \rightarrow \{4\})$
 - 表示从程序开头到当前节点的具体执行的集合
 - 其中
 - a的定值位置在1或者3或未定值
 - b的定值位置在4或未定值
 - 如：
 - 程序从a=1, b未初始化, c=0的状态开始执行
 - 则会走1, 2, 5的路径
 - 最后a在1定值, b未定值

```
1.  a=100;
2.  if (c > 0)
3.      a = 200;
4.      b = a;
5.  return a;
```



可达定值分析-合并操作

- 对应集合的并，即
 - $(\text{甲} \sqcup \text{乙})(x) = \text{甲}(x) \cup \text{乙}(x)$
 - 如： $(a \rightarrow \{1,2\}, b \rightarrow \{\}) \sqcup (a \rightarrow \{1,3\}, b \rightarrow \{4\})$
 $= (a \rightarrow \{1,2,3\}, b \rightarrow \{4\})$
- 正确性： 甲或者乙所包含的任意具体执行过程仍然包括在 $(\text{甲} \sqcup \text{乙})$ 中



可达定值分析- 节点转换函数

- $f_v(\text{甲})(x) = (\text{甲}(x) - \text{KILL}_v^x) \cup \text{GEN}_v^x$
 - 对于赋值语句 $x = \dots$
 - $\text{KILL}_v^x = \{\text{所有赋值给 } x \text{ 的语句编号}\}$
 - $\text{GEN}_v^x = \{\text{当前语句编号}\}$
 - 对于其他语句
 - $\text{KILL}_v^x = \text{GEN}_v^x = \{\}$
- 正确性：输入包含的具体执行序列加下一步仍然包括在输出中
- $f(\text{甲}) = (\text{甲} - \text{KILL}) \cup \text{GEN}$ 又叫数据流分析的标准型



数据流分析实现算法

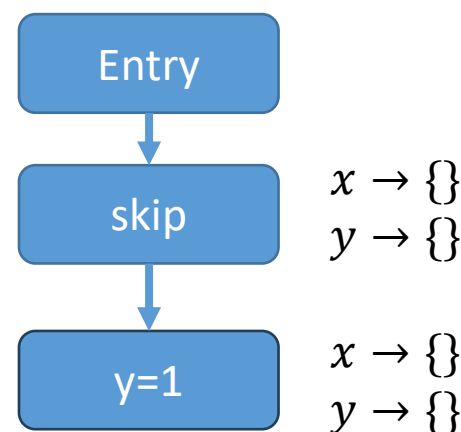
```
OUTentry = I
 $\forall v \in (V - \text{entry}): \text{OUT}_v \leftarrow \perp$ 
ToVisit  $\leftarrow V - \text{entry}$ 
While (ToVisit.size > 0) {
  v  $\leftarrow$  ToVisit中任意节点
  ToVisit -= v
   $\text{IN}_v \leftarrow \sqcup_{w \in \text{pred}(v)} \text{OUT}_w$ 
  If ( $\text{OUT}_v \neq f_v(\text{IN}_v)$ ) ToVisit  $\cup = \text{succ}(v)$ 
   $\text{OUT}_v \leftarrow f_v(\text{IN}_v)$ 
}
```

为什么？



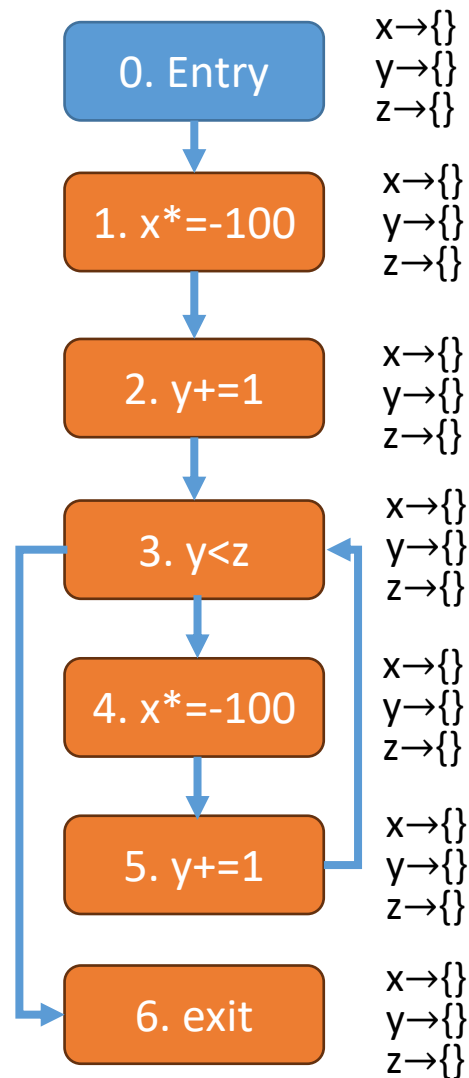
数据流分析实现算法

- 如果某个初始值在更新过程中没有被改变，后续节点也不会被更新
- 符号分析不存在这个问题，因为初始值不变意味着对应节点是死代码
- 可达定值分析也可以改成这种方式
 - 方式一：一开始就假设所有变量被entry定值
 - 方式二：设置特殊的抽象值作为初值
- 由于通用性更强，上页代码是数据流分析的标准代码
 - 之后会看到其他需要加入所有节点的情况
 - 通过特定选择顺序可以部分避免无效更新，如LIFO



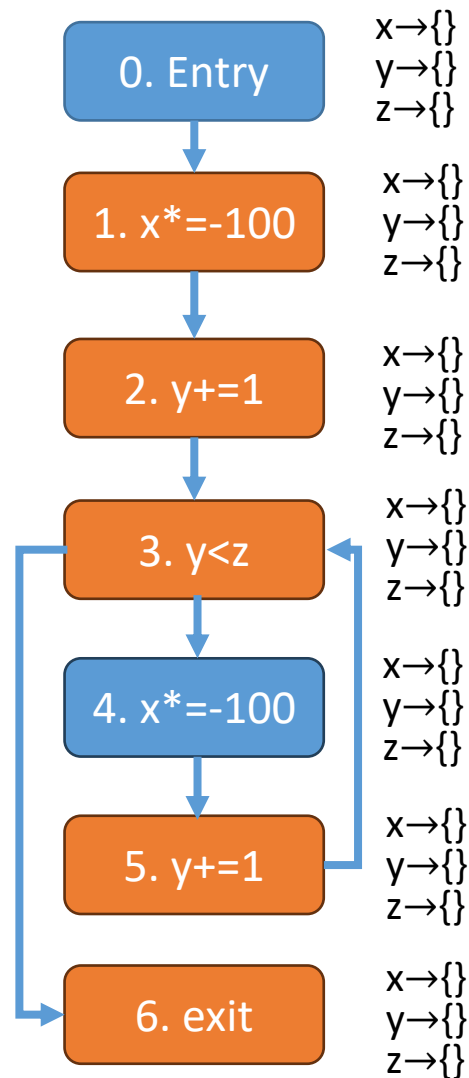


可达定值分析



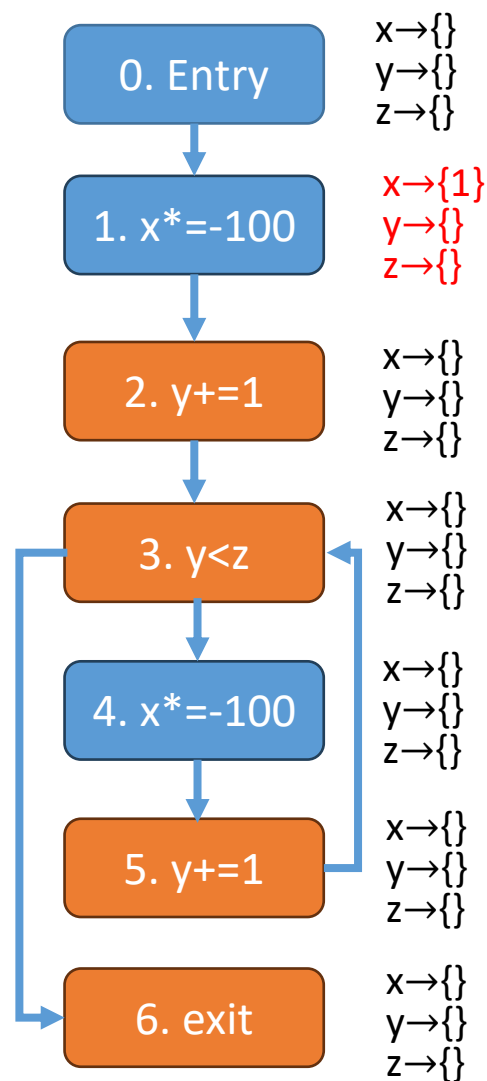


可达定值分析



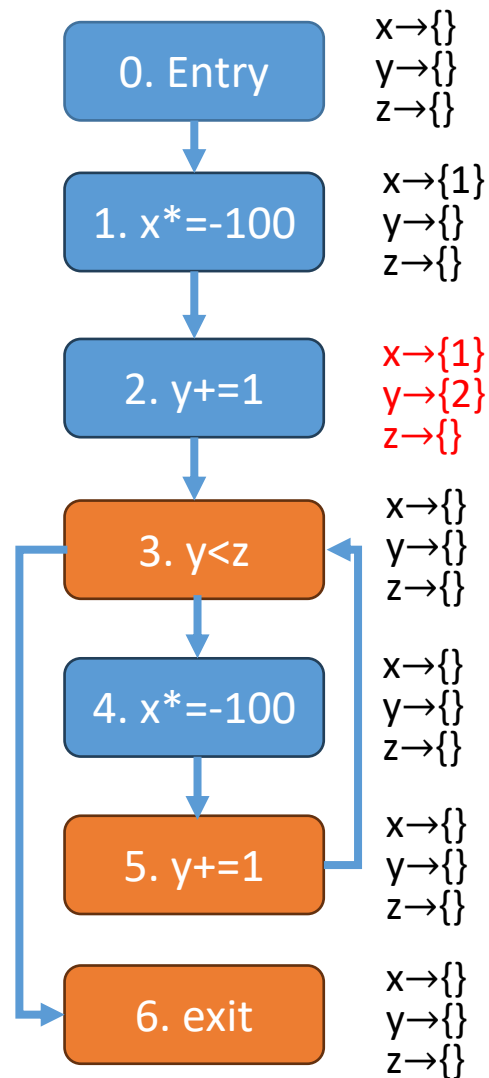


可达定值分析



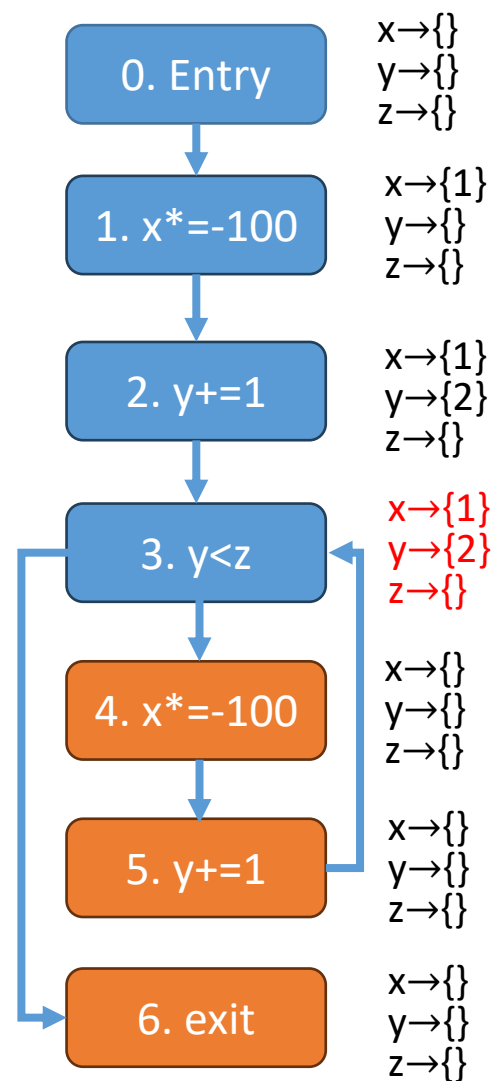


可达定值分析



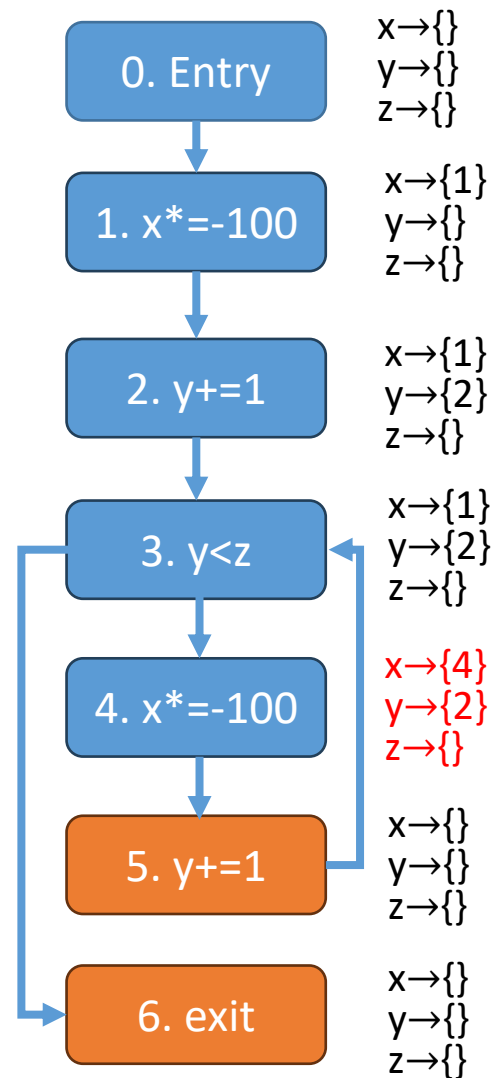


可达定值分析



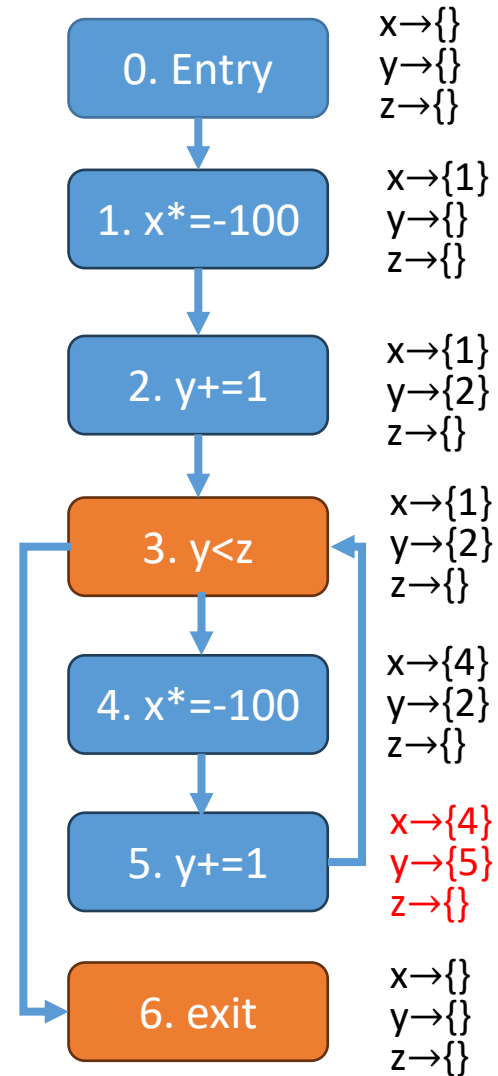


可达定值分析



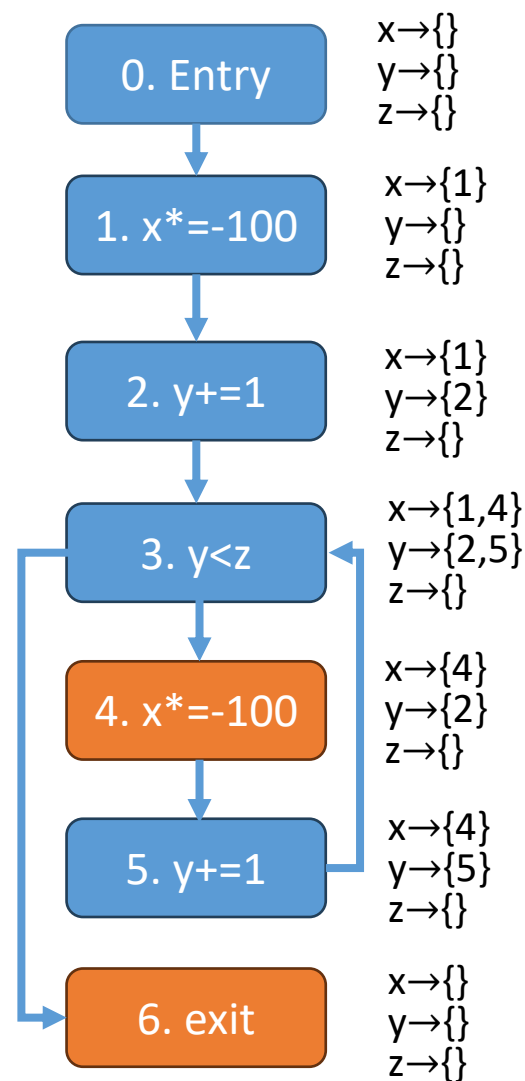


可达定值分析



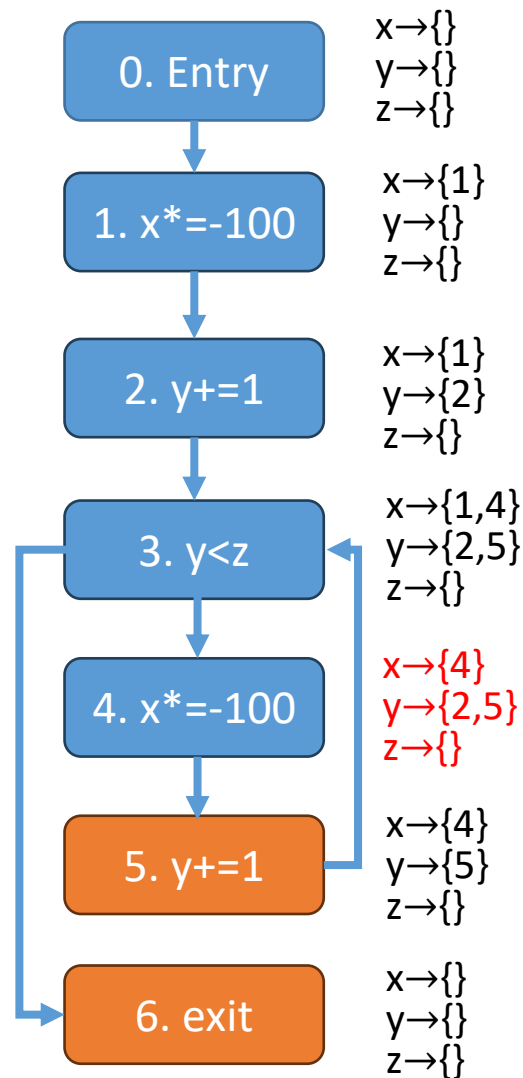


可达定值分析



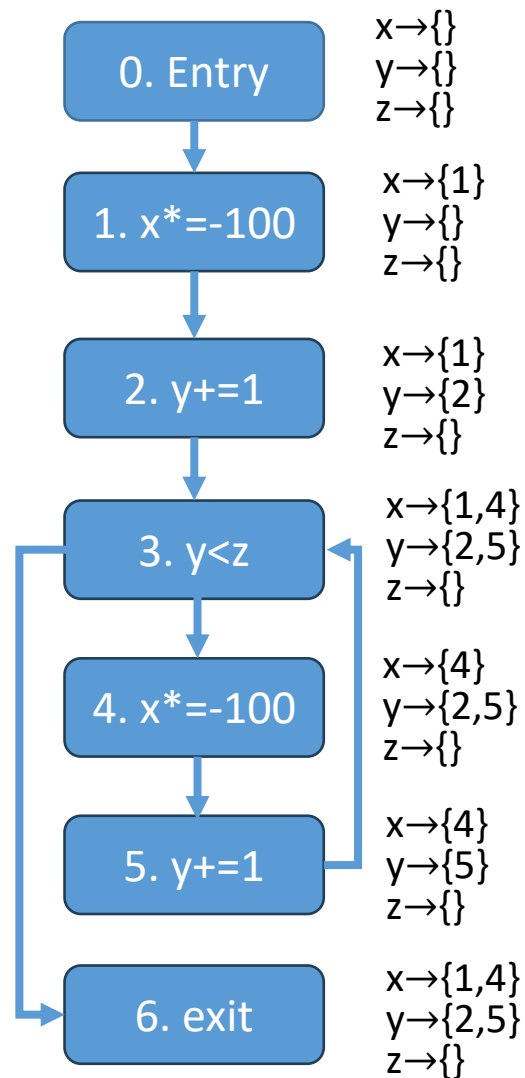


可达定值分析





可达定值分析





可达定值分析正确性

- 给定从头开始的任意具体执行序列，结束状态对应的变量定值位置都在抽象值的范围内
- 证明：在具体执行序列的长度上做归纳



可达定值分析——小结

- 和符号分析的关键不同点
 - 抽象值对应具体执行序列的集合
 - 在分析开头加入所有其他节点到ToVisit
- 由于更通用，标准数据流分析也按照以上两种情况定义



可用表达式 (available expression) 分析

- 给定程序中某个位置p，如果从入口到p的所有路径都对表达式exp求值，并且最后一次求值后该表达式的所有变量都没有被修改，则exp称作p的一个可用表达式。给出分析寻找可用表达式。
 - 要求下近似
 - 例：

1. <code>a=c+(b+10);</code>	1运行结束的时候可用表达式是b+10、c+(b+10)
2. <code>if (a>b)</code>	2运行结束的时候可用表达式是b+10、c+(b+10)
3. <code>c = a+10;</code>	3运行结束的时候可用表达式是b+10、a+10
4. <code>return a;</code>	4运行结束的时候可用表达式是b+10



可用表达式分析——抽象域

- 抽象值：表达式的集合
- 初始值：程序中表达式的全集
- 抽象值含义：对应从头到当前位置的具体执行序列的集合，其中可用表达式是当前位置抽象值的超集



可用表达式分析——合并

- 集合交
- 正确性：甲或者乙所包含的任意具体执行过程仍然包括在 $(甲 \sqcup 乙)$ 中

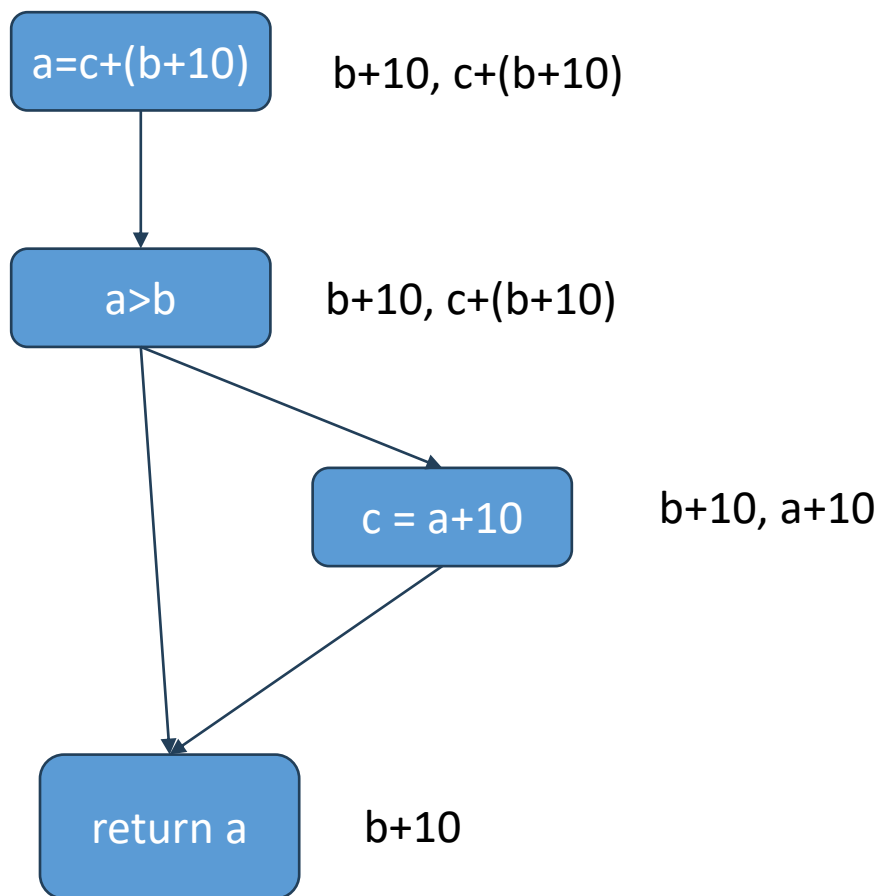


可用表达式分析- 节点转换函数

- $f_v(\text{甲}) = (\text{甲} - \text{KILL}_v) \cup \text{GEN}_v$
 - 对于赋值语句 $x = \dots$
 - $\text{KILL}_v = \{\text{所有包含 } x \text{ 的表达式}\}$
 - $\text{GEN}_v = \{\text{当前语句中求值的不含 } x \text{ 的表达式}\}$
 - 对于其他语句
 - $\text{KILL}_v = \{\}$
 - $\text{GEN}_v = \{\text{当前语句中求值的表达式}\}$
- 正确性：输入包含的具体执行序列加下一步仍然包括在输出中



可用表达式分析——例子





可用表达式分析正确性

- 给定从头开始的任意具体执行序列，结束状态对应的可用表达式是抽象值的超集
- 证明：在具体执行序列的长度上做归纳



可用表达式分析-小结

- 和之前的分析不同
 - 分析是下近似，即抽象域包含的为具体域的子集
 - 初始值为全集
 - 合并操作为求交
- 标准数据流框架同时提供对上近似和下近似的支持



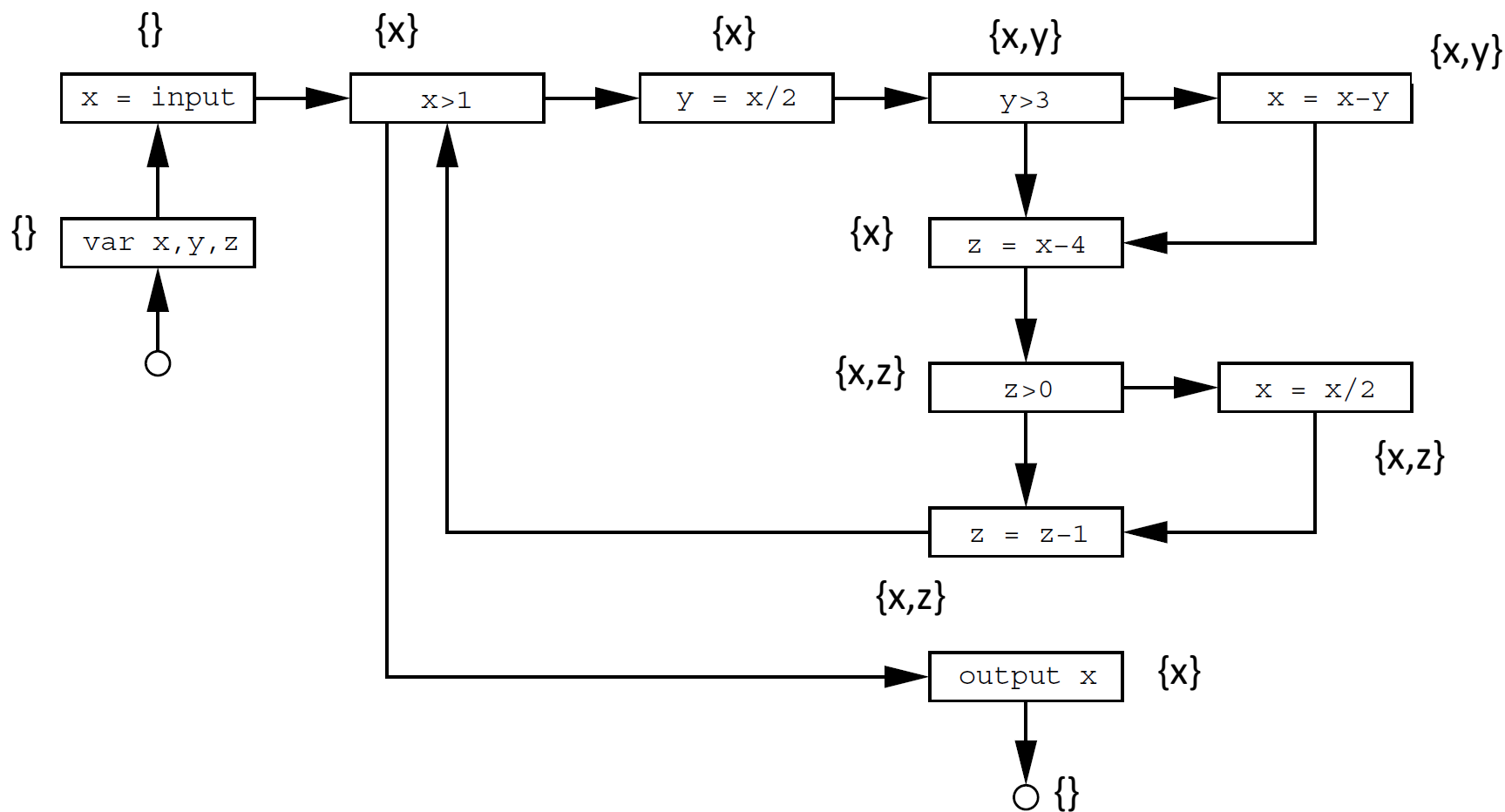
活跃变量分析 (Liveness Analysis)

- 活跃变量：给定程序中的某条语句 s 和变量 v ，如果在 s 执行前保存在 v 中的值在后续执行中还会被读取就被称作活跃变量
- 第四行的 x 和 y 是否为活跃变量？
 - x 活跃， y 不活跃
- 第八行的 x 和 z 呢？
 - x 和 z 都活跃
- 活跃变量分析：返回所有可能的活跃变量
 - 上近似

```
1.  var x,y,z;
2.  x = input;
3.  while (x>1) {
4.      y = x/2;
5.      if (y>3) x = x-y;
6.      z = x-4;
7.      if (z>0) x = x/2;
8.      z = z-1;
9.  }
10. output x;
```



活跃变量分析-例子





活跃变量分析——抽象域

- 抽象值：变量的集合
- 初始值：空集
- 抽象值含义：对应从当前位置开始的任意长度的具体执行序列的集合
 - 因为程序有可能会无限执行，不能定义为到exit结束的序列



活跃变量分析——合并

- 集合并
- 正确性：甲或者乙所包含的任意具体执行过程仍然包括在 $(甲 \sqcup 乙)$ 中



活跃变量分析- 节点转换函数

- $f_v(\text{甲}) = (\text{甲} \setminus \text{KILL}_v) \cup \text{GEN}_v$
 - $\text{GEN}_v = v$ 中读取的所有变量
 - $\text{KILL}_v = \begin{cases} \{x\} & v := x = \text{exp}; \\ \{x\} & v := \text{int } x; \\ \{\} & \text{otherwise} \end{cases}$
- 正确性：输入包含的具体执行序列加上一步仍然包括在输出中



活跃变量分析实现算法

```
INexit = I
 $\forall v \in (V - \text{exit}): \text{IN}_v \leftarrow \top$ 
ToVisit  $\leftarrow V - \text{exit}$ 
While (ToVisit.size > 0) {
  v  $\leftarrow$  ToVisit中任意节点
  ToVisit -= v
   $\text{OUT}_v \leftarrow \sqcup_{w \in \text{succ}(v)} \text{IN}_w$ 
  If ( $\text{IN}_v \neq f_v(\text{OUT}_v)$ ) ToVisit  $\cup = \text{pred}(v)$ 
   $\text{IN}_v \leftarrow f_v(\text{OUT}_v)$ 
}
```

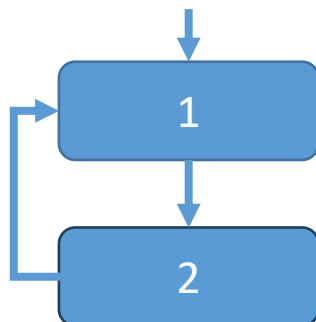
如果引入单独的初值，
是否还需要加入所有
节点

交换IN和OUT，交换pred和succ，交换exit和entry
等价于把数据流的所有箭头方向交换，exit和entry后应用原算法



活跃变量分析实现算法

- 需要，可能会有不能到达exit的执行序列





活跃变量分析-正确性

- 和之前的问题不同，因为序列不一定在exit结束，所以不能从exit出发反向做归纳
- 给定任意长度的具体执行序列，以开始位置看，在该序列上活跃的变量都在开始位置的抽象值中
- 证明：从序列的结束位置开始反向对长度做归纳
- 该形式也是更通用的形式



活跃变量分析一小结

- 和之前的分析不同
 - 分析是反向开始
 - 程序有可能不结束，也就是说执行序列不一定到达exit
 - 和之前执行序列一定从entry开始不同
 - 需要一开始将所有exit之外节点加入toVisit
 - 正确性要从一段任意长度的执行序列定义



作业：繁忙表达式分析 (very busy expression)

- 繁忙表达式：从执行某个程序节点之前开始，在其中变量被修改之前，一定会被读取的表达式
- 繁忙表达式分析：找到每个程序节点的繁忙表达式
 - 要求下近似
 - 如：
 1. if ($a > b$)
 2. $x = b - a$
 3. $y = x - y + (a + b + b)$
 4. else
 5. $y = b - a$
 6. $x = x - y + (a + b)$
 - 在第一行， $b - a$, $a + b$, $a > b$ 为繁忙表达式



参考资料

- 《编译原理》 第9章
- Lecture Notes on Static Analysis
 - <https://cs.au.dk/~amoeller/spa/>