软件分析

数据流分析：示例

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}
舅={所有的整数}

计算过程	具体值范围	抽象值	抽象值范围
x	[1, +∞]	正	[1, +∞]
x+1	[2, +∞]	正	[1, +∞]
x+1-1	[1, +∞]	舅	[−∞, +∞]
y	[1, +∞]	正	[1, +∞]
x+1-1+y	[2, +∞]	舅	[−∞, +∞]

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
  - γ(正) ={x∈ 整数 | $x > 0$}
- α将具体域的值映射到抽象域
  - $\alpha(1)$ = 正
  - 需要满足： $x \in \gamma(\alpha(x))$
- 运算操作+, -, ×分别定义抽象域上的版本⊕,⊖,⊗
  - 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： $\forall x \in \gamma(甲) \wedge y \in \gamma(乙),\ x + y \in \gamma(甲 \oplus 乙)$

⊕	正	负	零	槑
正	正			
负	槑	负		
零	正	负	零	
槑	槑	槑	槑	槑

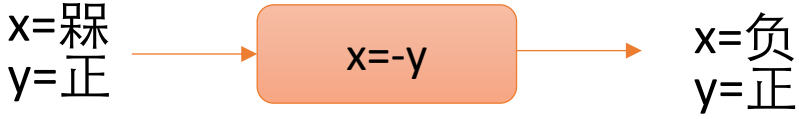
程序 vs 表达式

- 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
- ……

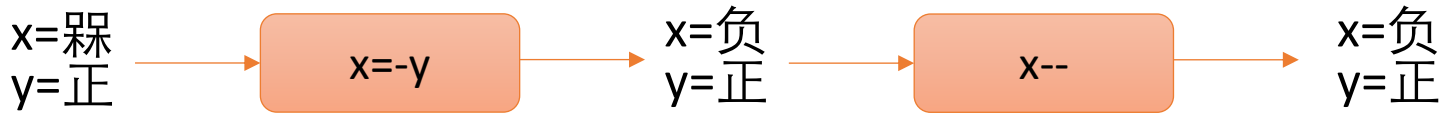
表示x为负数，y为正数的所有系统状态的集合

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

正确性：输入抽象状态包含的任意具体状态的执行结果在输出抽象状态中

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？



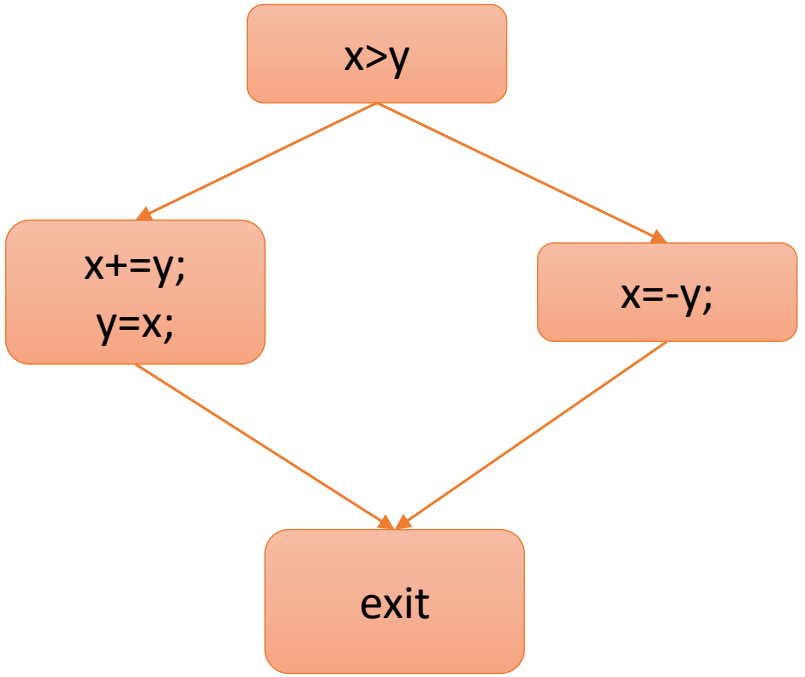
正确性：输入抽象状态包含的任意具体状态的执行结果在输出抽象状态中

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



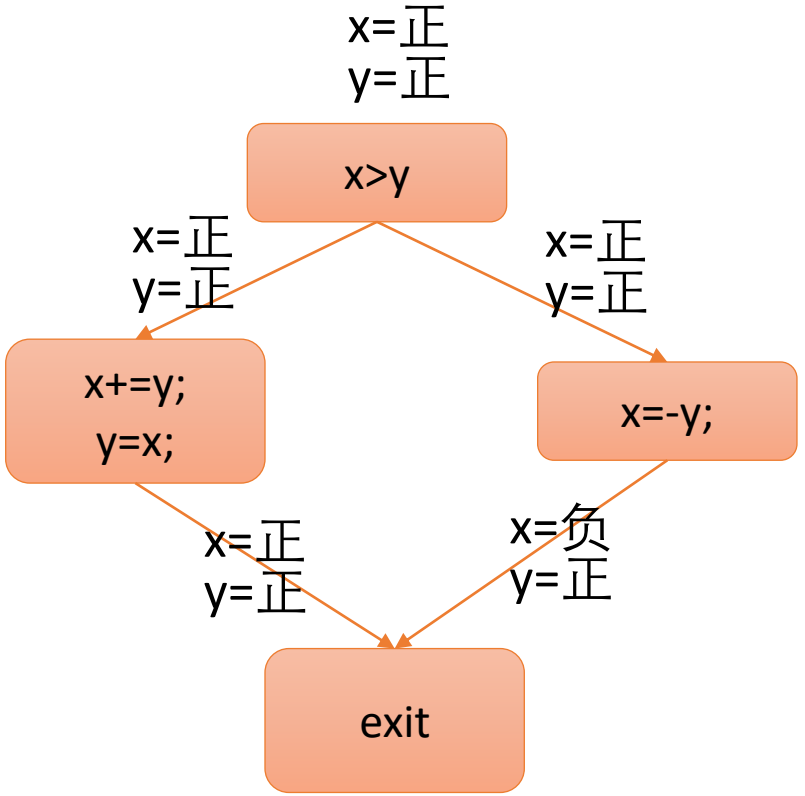
输入抽象状态对应具体状态左右路径都有可能走

分支

- 例

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

- 输入：
  - x=正
  - y=正
- 求输出的符号



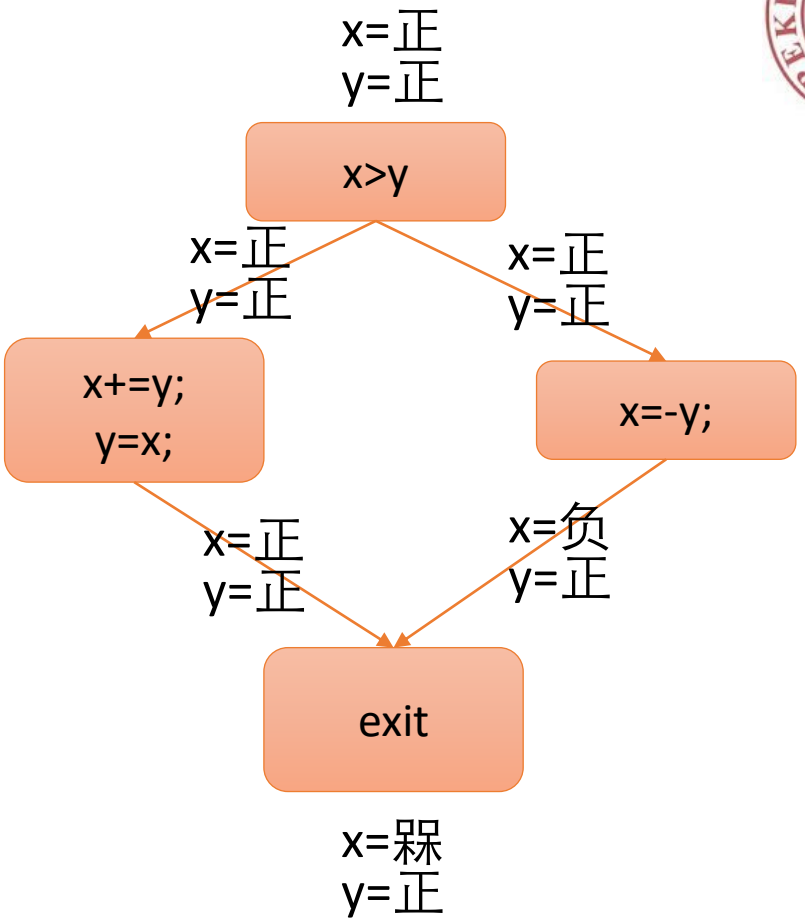
针对两条路径分别计算

分支

- 例

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

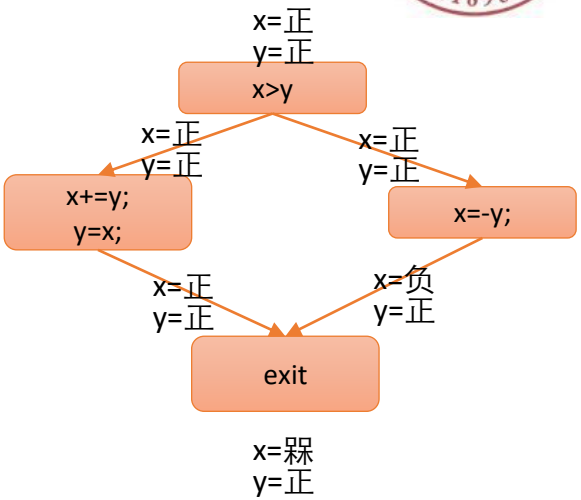
- 输入：
  - x=正
  - y=正
- 求输出的符号



定义合并操作：

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

分支分析正确性



- 输入抽象状态包含的任意具体状态的执行结束状态在输出抽象状态中
- 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. 任一分支一定保证执行结束状态在输出抽象值中
4. 合并时， ⊔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⊔满足交换律

循环

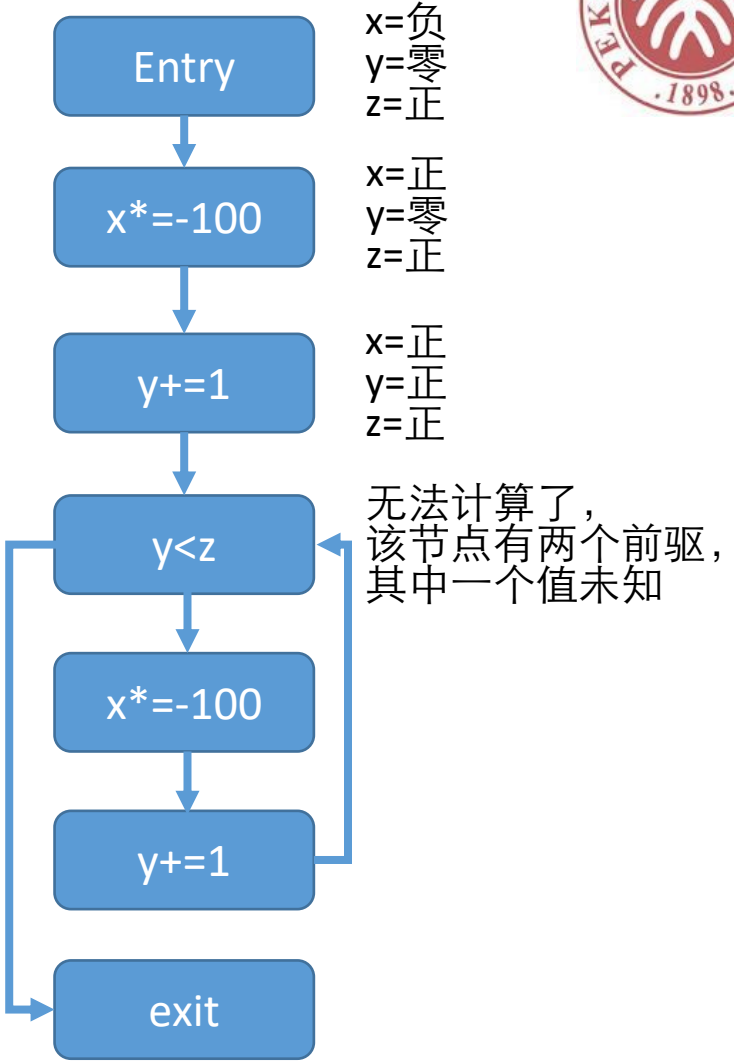
```
x*=-100;
y+=1;
while(y < z) {
    x *= -100;
    y += 1;
}
```

输入：x为负，y为零，z为正

输出：x为踩，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

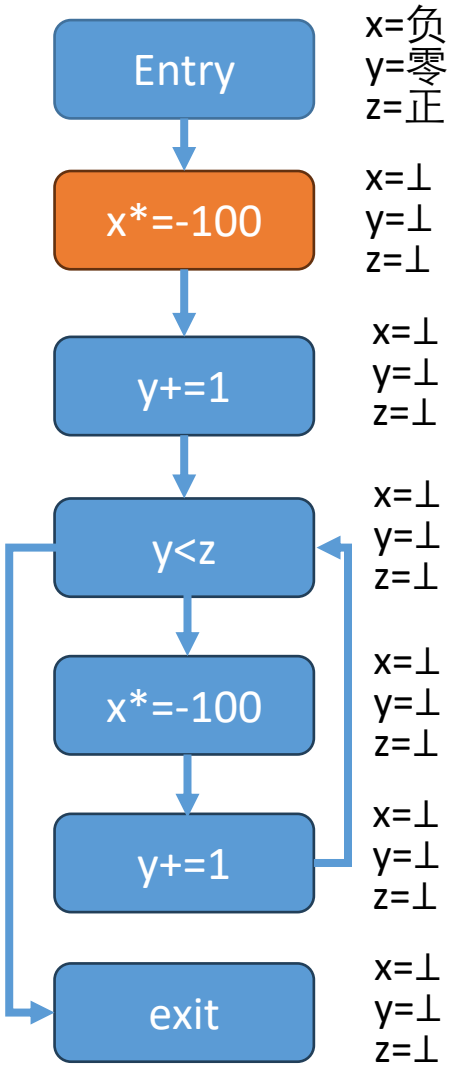


循环——解决方案

- 引入抽象值⊥
  - $\gamma(\perp) = \emptyset$
- ⊔操作扩展到⊥：$x \sqcup \perp = x$
- 初始节点上的抽象值都为⊥，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

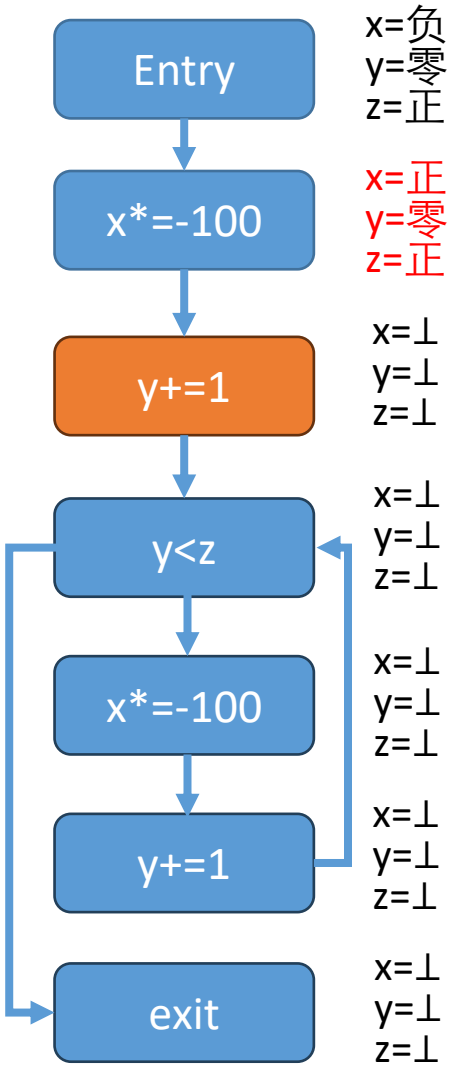
循环

- 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



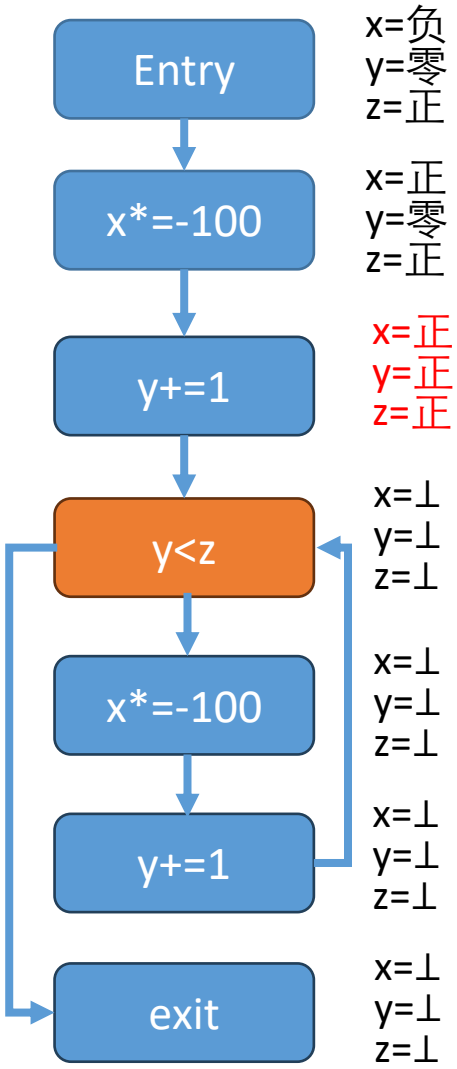
循环

- 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



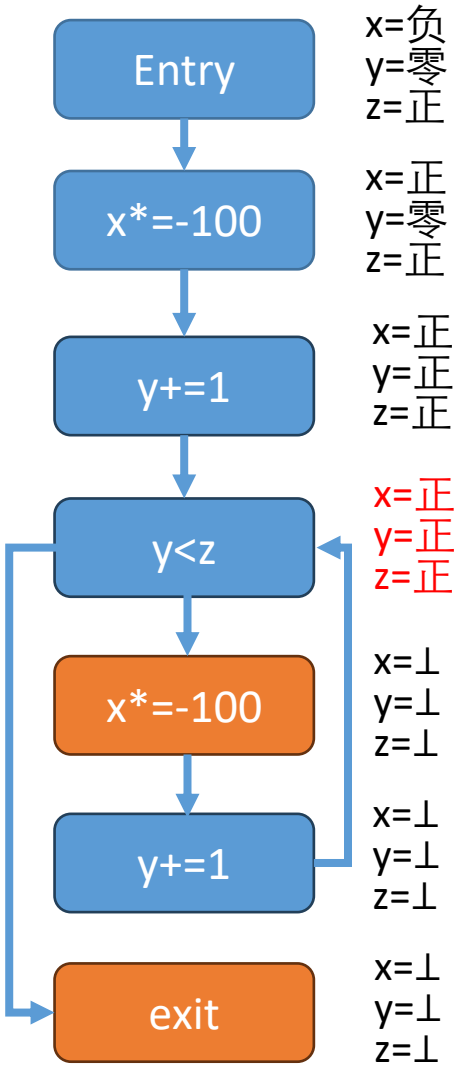
循环

- 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



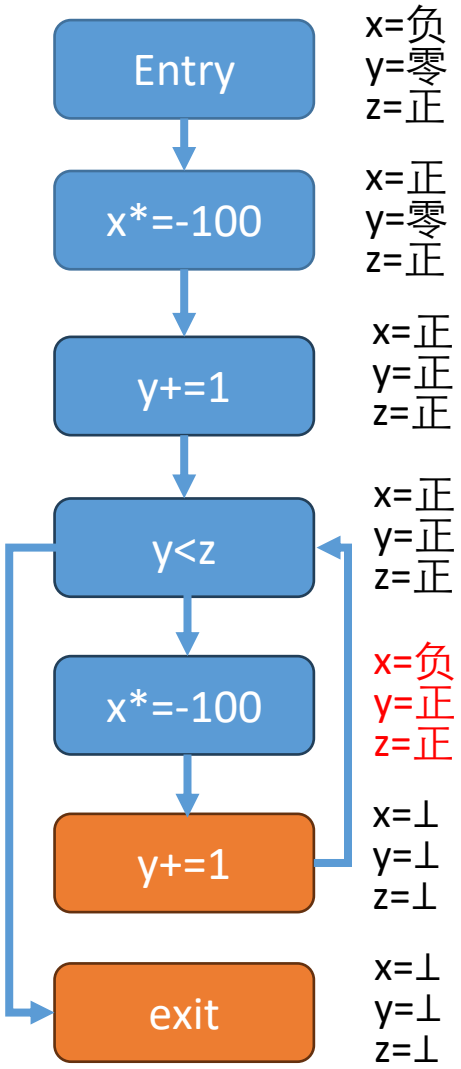
循环

- 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



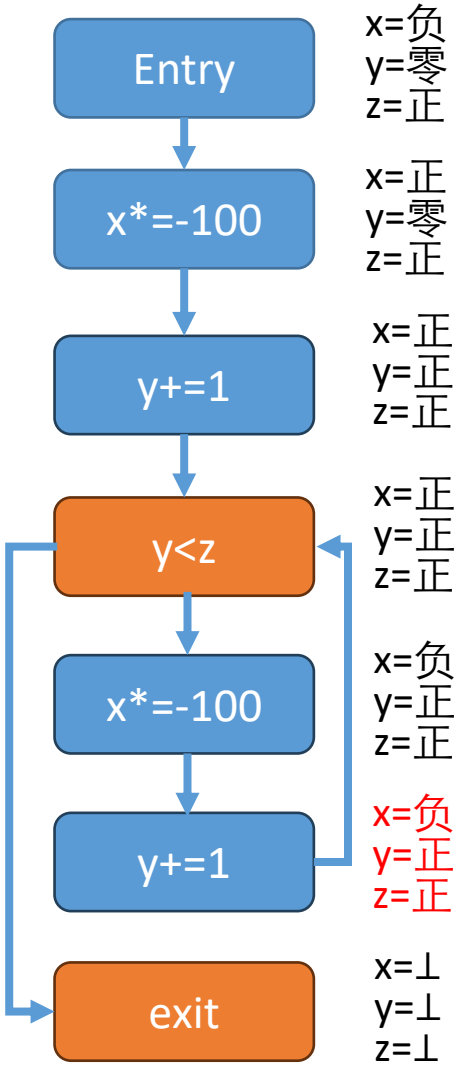
循环

- 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



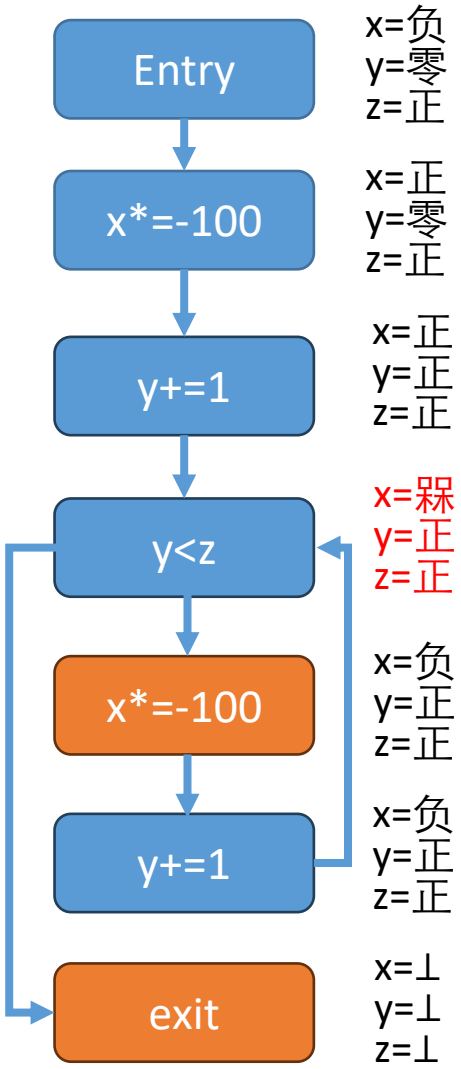
循环

- 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



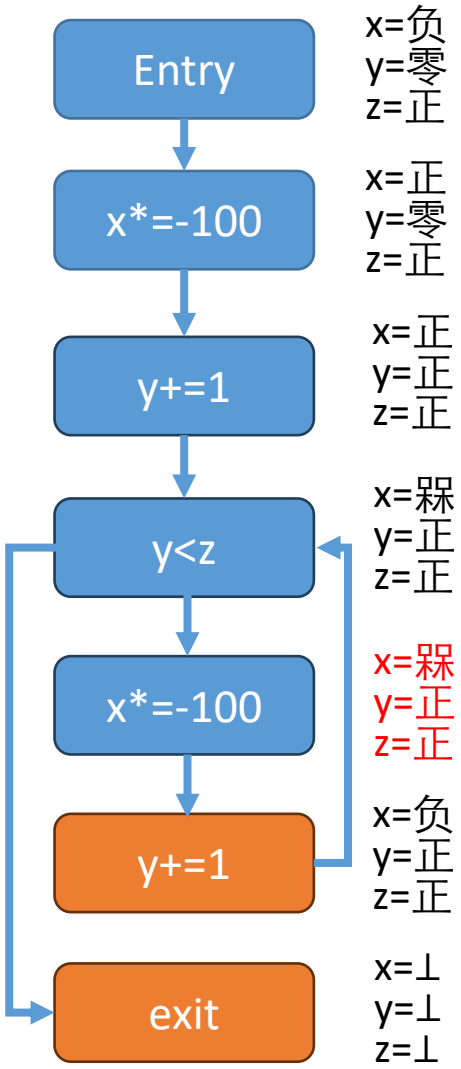
循环

- 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



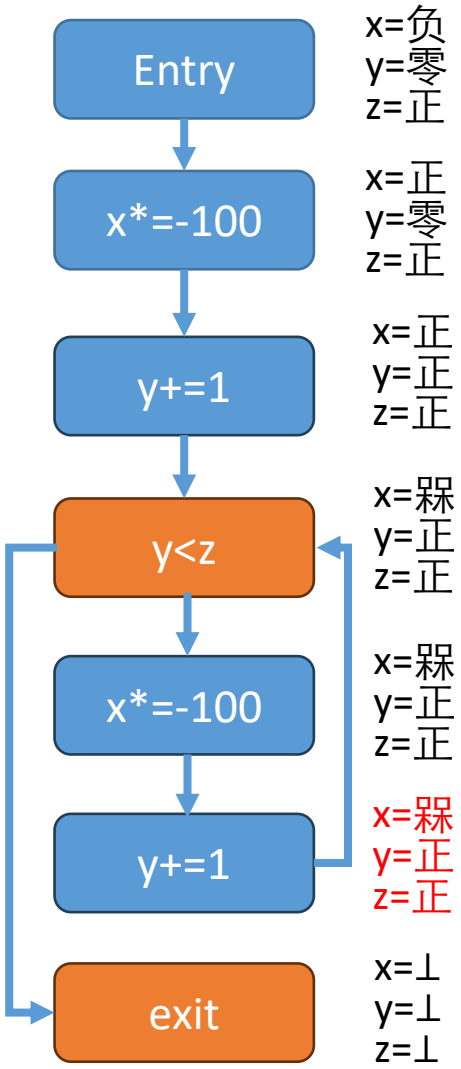
循环

- 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



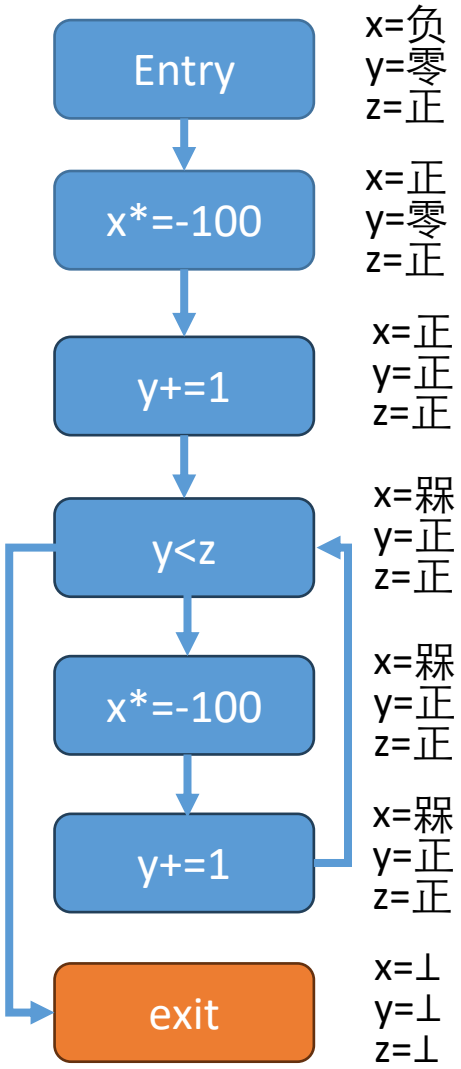
循环

- 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



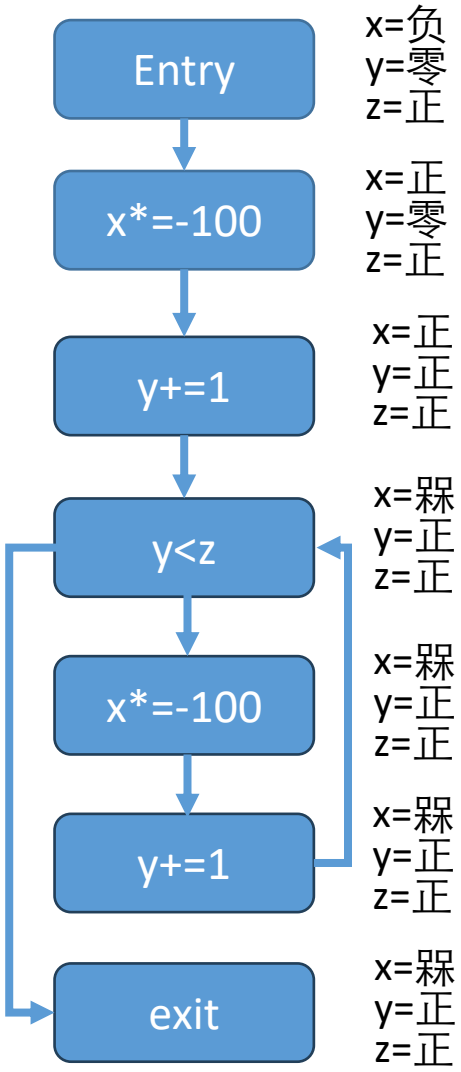
循环

- 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

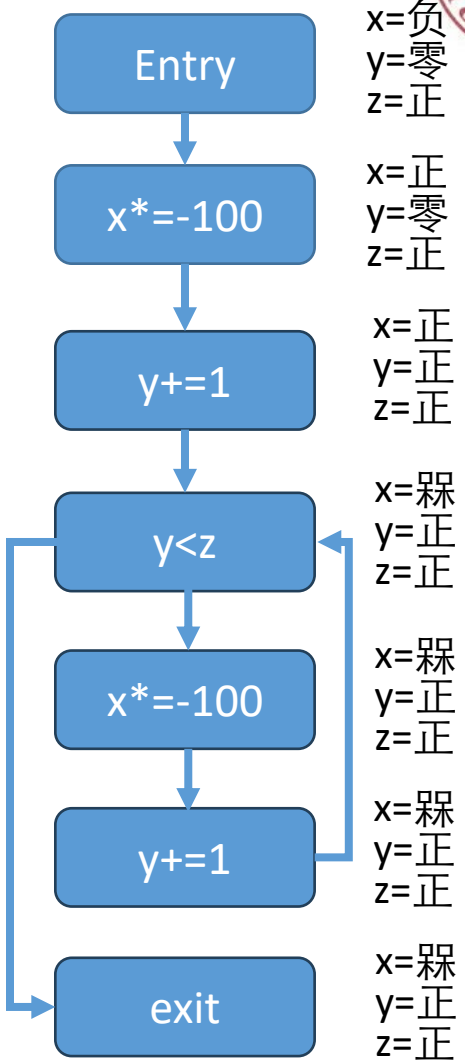
- 令$\mathbf{S} = \{(s_x, s_y, s_z) | s_x, s_y, s_z \in \{正, 负, 零, 踩, \bot\}\}$
- 每个节点的值为**S**的一个元素，代表对应语句执行之后的抽象状态，用OUT表示
  - 同时用IN表示语句执行之前的抽象状态
- 初始值
  - OUT_{entry}=(负,零,正)
  - $\mathrm{OUT}_{其他节点}$=($\bot, \bot, \bot$)
- 节点转换函数$f_v: \boldsymbol{S} \to \boldsymbol{S}$
  - $f_{exit} = id$
  - $f_{其他节点}$=根据相应语句进行计算
- 合并运算$\mathrm{IN}_v = \sqcup_{w \in pred(v)} \mathrm{OUT}_w$
- 节点更新运算$\mathrm{OUT}_v = f_v(\mathrm{IN}_v)$
- 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

$OUT_{entry} = I$
$\forall v \in (V - entry): OUT_v \leftarrow \top$
ToVisit $\leftarrow$ succ(entry)
While(ToVisit.size > 0) {
  $v \leftarrow$ ToVisit中任意节点
  ToVisit -= v
  $IN_v \leftarrow \sqcup_{w \in pred(v)} OUT_w$
  If$\left(OUT_v \neq f_v(IN_v)\right)$ ToVisit $\cup=$ succ(v)
  $OUT_v \leftarrow f_v(IN_v)$
}

分析为什么是对的?

- 因为计算到收敛，所以分析结束的时候下面这些等式必然成立的
  - OUT_{entry}=(负,零,正)
  - $\mathrm{IN}_v = \sqcup_{w \in pred(v)} \mathrm{DATA}_w$
  - $\mathrm{OUT}_v = f_v(\mathrm{IN}_v)$
  - 其中v为entry之外的任意节点



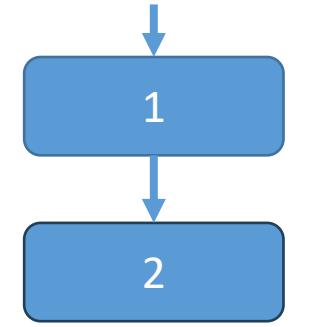
符号分析正确性

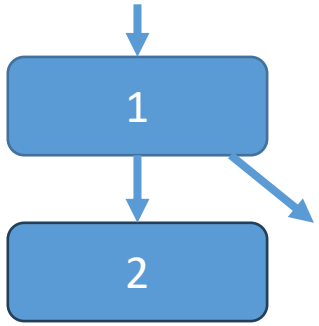
- 如果分析正常结束，给定输入抽象值对应的任意具体输入，程序执行到任意节点的状态一定包含在该节点的抽象值中。
- 证明和分支的情况类似：
  - 该具体输入的执行产生一条路径。在路径长度上做归纳，证明路径结束位置的抽象值包括具体值。
  - 路径长度为1的时候，显然成立。

  - 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

符号分析正确性

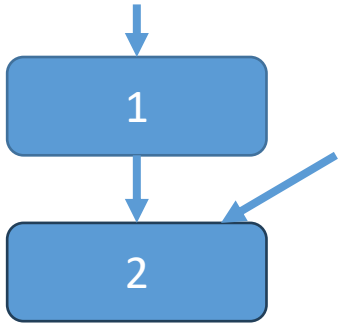
- 情况1：路径末尾只有一个后继节点，没有分叉和合并
  - $\mathrm{OUT}_2 = f_2(\mathrm{OUT}_1)$
  - 转换函数正确时，该情形保证正确
- 情况2：路径末尾有分叉的情况
  - $\mathrm{OUT}_2 = f_2(\mathrm{OUT}_1)$
  - 同上，仍然正确





符号分析正确性

- 情况3：路径末尾有合并的情况
  - $\mathrm{OUT}_2 = f_2(\mathrm{OUT}_1 \sqcup \text{其他值})$
  - $\sqcup$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

- 采用四套近似方案处理程序
- 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⊔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=踩

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=正
- 输出：x=正
  - 额外引入了x=1的不精确情况

近似方案3的不精确

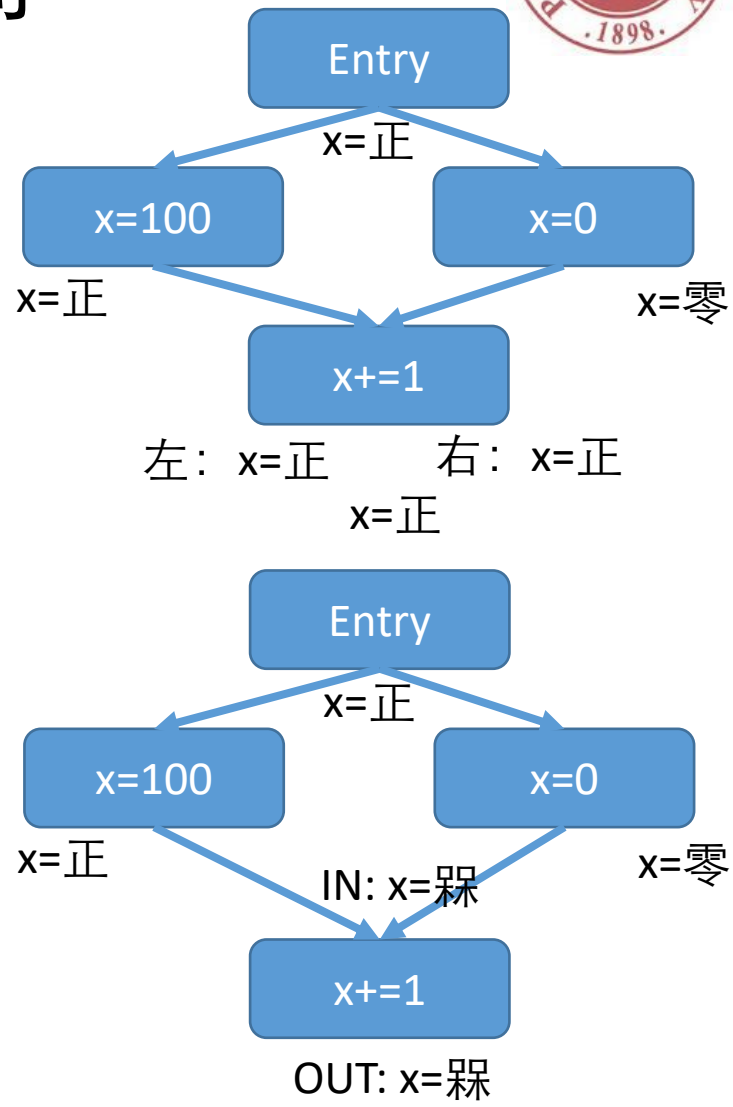
- 近似方案3：在控制流路径分叉时，复制抽象状态到所有分支
- 每个具体状态只能到达一个分支，形成不精确

x=踝
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=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

近似方案4的不精确

- 近似方案4：在控制流路径合并时，用⊔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（之后介绍）
- 近似方案4：在控制流路径合并时，用⊔操作合并多个抽象状态
  - 设计⊔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.  a=100;
2.  if (c > 0)
3.    a = 200;
4.    b = a;
5.  return a;
```

运行到1的时候a的定值是1
运行到2的时候a的定值是1
运行到3的时候a的定值是3
运行到4的时候a的定值是3， b的定值是4
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：从变量到语句标号集合的映射
  - a->{1}, b->{}
  - a->{1, 3}, b->{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
  - γ(a−>{1, 3}, b−>{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
  - $(\text{甲} \sqcup \text{乙})(\text{x}) = \text{甲}(x) \cup \text{乙}(x)$
  - 如：$(a \to \{1,2\}, b \to \{\}) \sqcup (a \to \{1,3\}, b \to \{4\})$
    $= (a \to \{1,2,3\}, b \to \{4\})$
- 正确性：甲或者乙所包含的任意具体执行过程仍然包括在(甲 ⊔ 乙)中

可达定值分析-节点转换函数

- $f_v(\text{甲})(x) = (\text{甲}(x) - \text{KILL}_v^x) \cup GEN_v^x$
  - 对于赋值语句x=…
    - KILL_v^x ={所有赋值给x的语句编号}
    - $\text{GEN}_\text{v}^\text{x}$ ={当前语句编号}
  - 对于其他语句
    - KILL_v^x = $\text{GEN}_\text{v}^\text{x}$ ={}
- 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
- $f(\text{甲}) = (\text{甲} - \text{KILL}) \cup GEN$又叫数据流分析的标准型

数据流分析实现算法

$OUT_{entry} = I$

$\forall v \in (V - entry): OUT_v \leftarrow \top$

$ToVisit \leftarrow V - entry$

$While(ToVisit.size > 0)\ \{$

$\quad v \leftarrow$ ToVisit中任意节点

$\quad ToVisit\ \text{-}= v$

$\quad IN_v \leftarrow \sqcup_{w \in pred(v)} OUT_w$

$\quad If\left(OUT_v \neq f_v(IN_v)\right)\ ToVisit \cup= succ(v)$

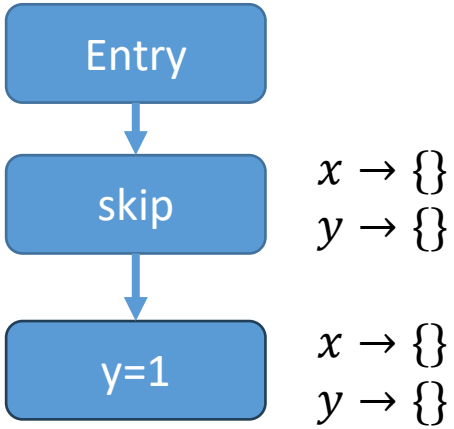
$\quad OUT_v \leftarrow f_v(IN_v)$

$\}$

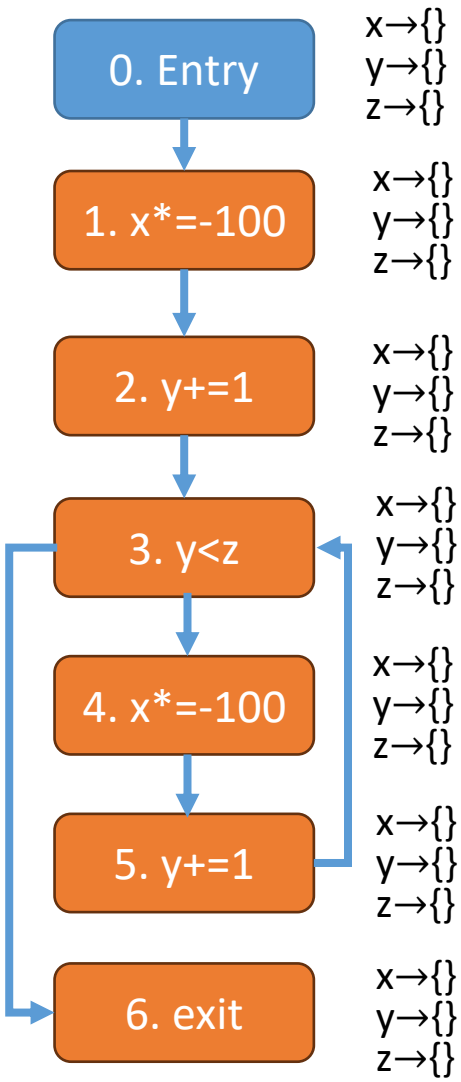
为什么？

数据流分析实现算法

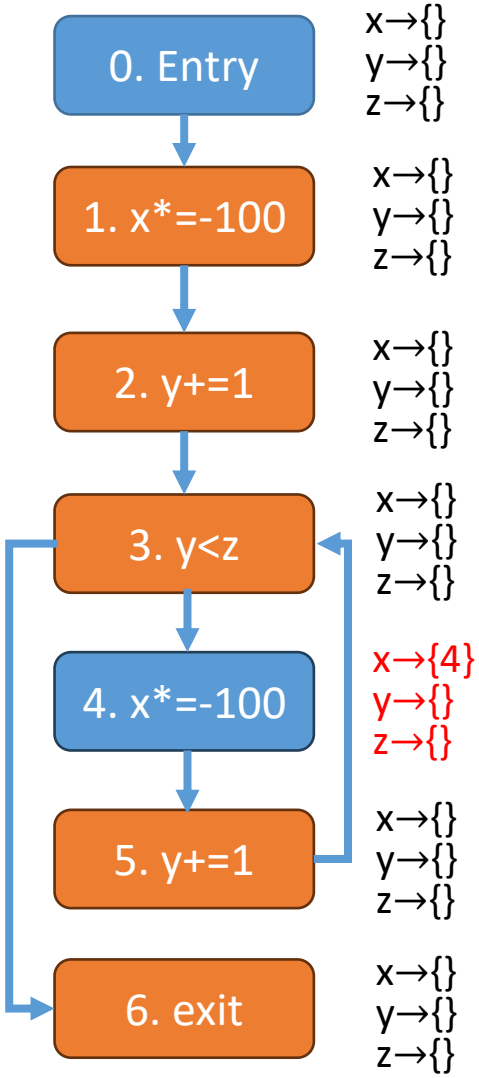
- 如果某个初始值在更新过程中没有被改变，后续节点也不会被更新
- 符号分析不存在这个问题，因为初始值不变意味着对应节点是死代码
- 可达定值分析也可以改成这种方式
  - 方式一：一开始就假设所有变量被entry定值
  - 方式二：设置特殊的抽象值作为初值
- 由于通用性更强，上页代码是数据流分析的标准代码
  - 之后会看到其他需要加入所有节点的情况
  - 通过特定选择顺序可以部分避免无效更新，如LIFO



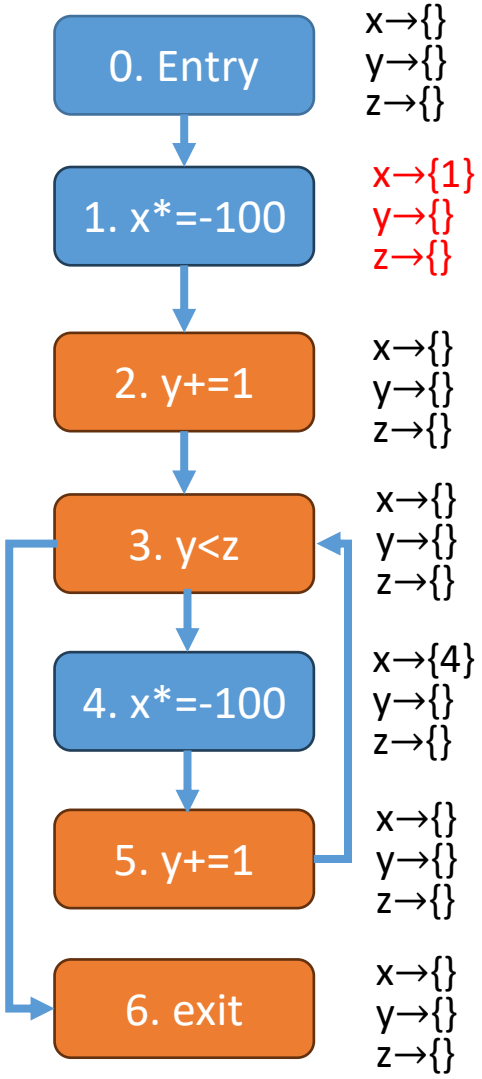
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{}
y→{}
z→{}
2. y+=1
x→{}
y→{}
z→{}
3. y<z
x→{}
y→{}
z→{}
4. x*=-100
x→{}
y→{}
z→{}
5. y+=1
x→{}
y→{}
z→{}
6. exit
x→{}
y→{}
z→{}

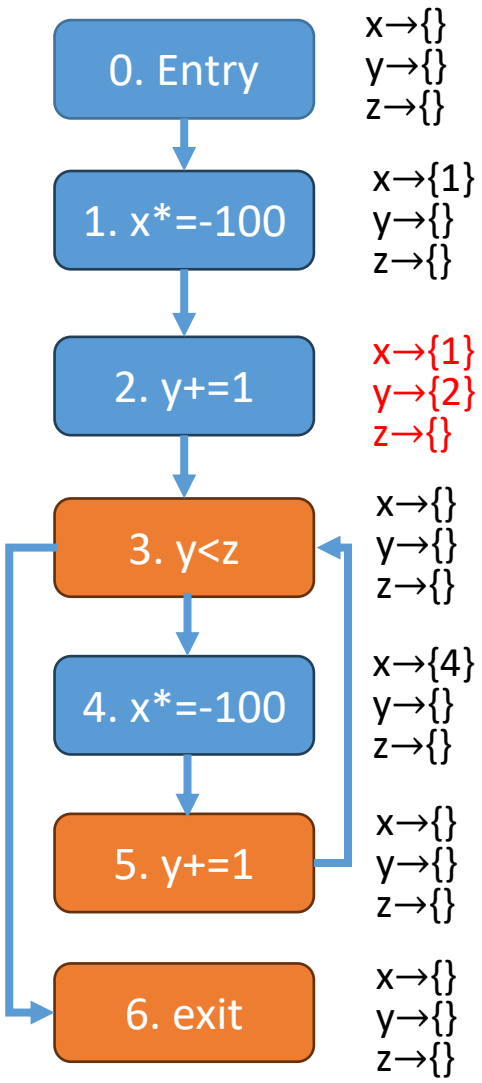
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{}
y→{}
z→{}
2. y+=1
x→{}
y→{}
z→{}
3. y<z
x→{}
y→{}
z→{}
4. x*=-100
x→{4}
y→{}
z→{}
5. y+=1
x→{}
y→{}
z→{}
6. exit
x→{}
y→{}
z→{}

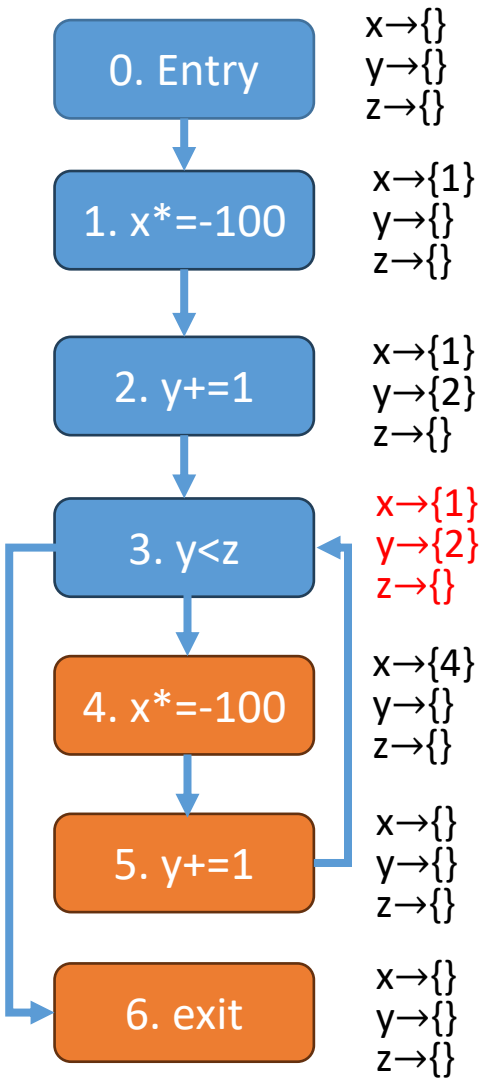
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{}
y→{}
z→{}
3. y<z
x→{}
y→{}
z→{}
4. x*=-100
x→{4}
y→{}
z→{}
5. y+=1
x→{}
y→{}
z→{}
6. exit
x→{}
y→{}
z→{}

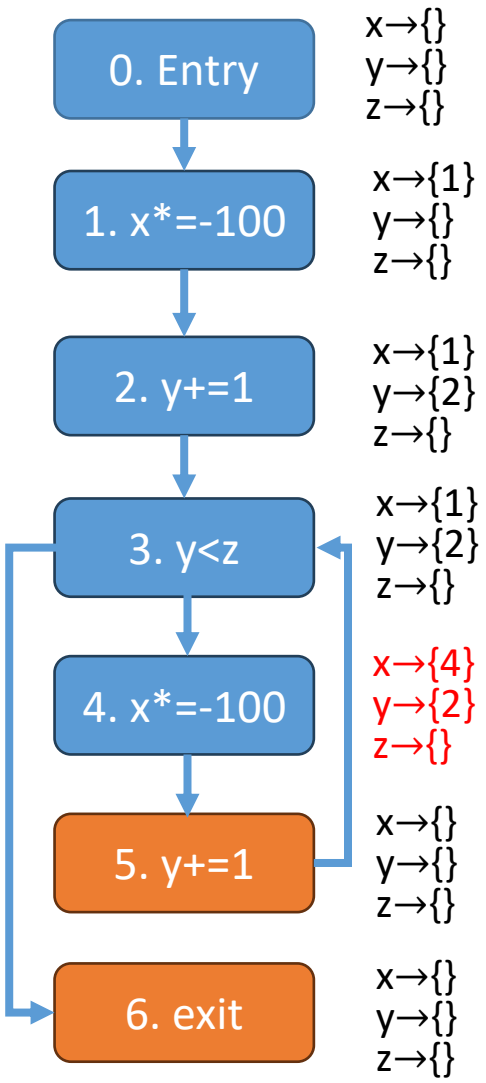
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{}
y→{}
z→{}
4. x*=-100
x→{4}
y→{}
z→{}
5. y+=1
x→{}
y→{}
z→{}
6. exit
x→{}
y→{}
z→{}

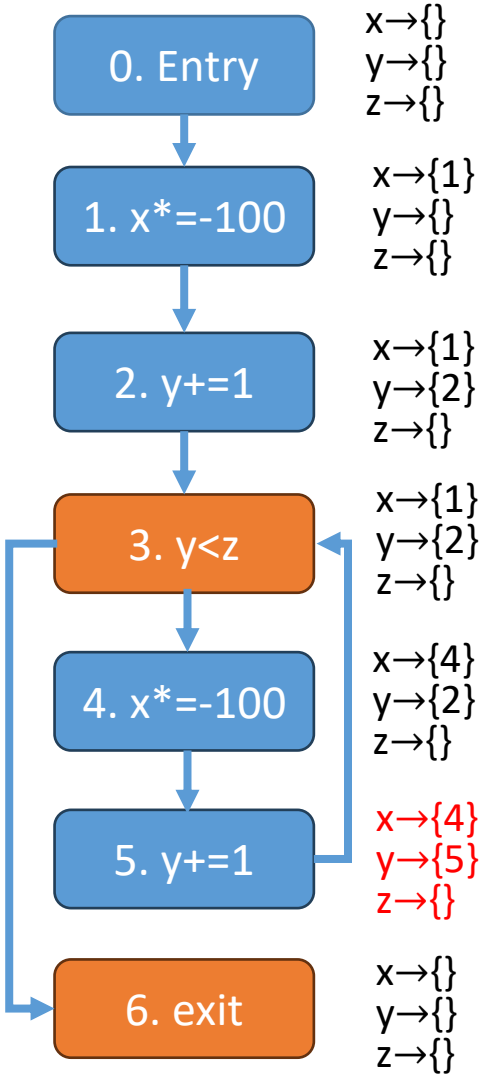
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{1}
y→{2}
z→{}
4. x*=-100
x→{4}
y→{}
z→{}
5. y+=1
x→{}
y→{}
z→{}
6. exit
x→{}
y→{}
z→{}

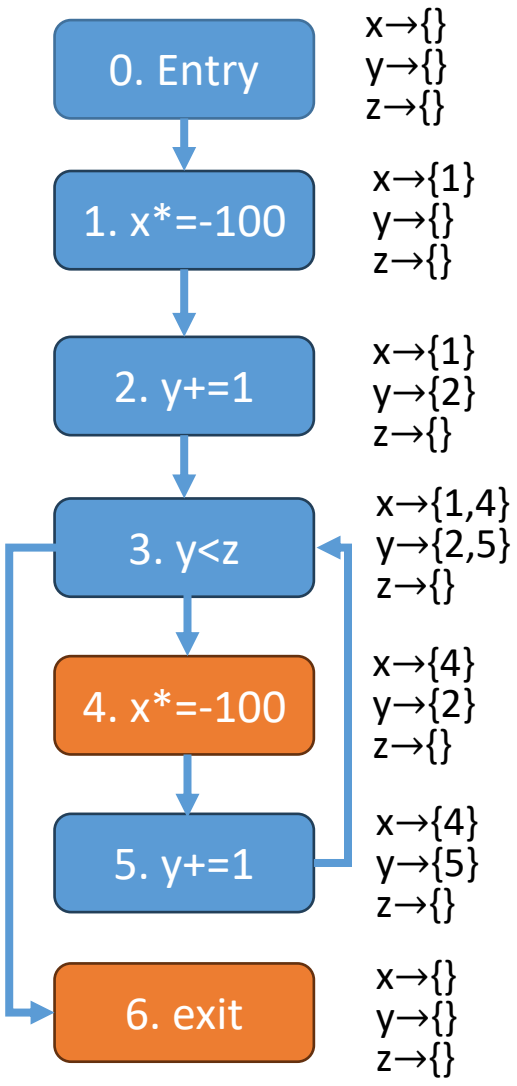
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{1}
y→{2}
z→{}
4. x*=-100
x→{4}
y→{2}
z→{}
5. y+=1
x→{}
y→{}
z→{}
6. exit
x→{}
y→{}
z→{}

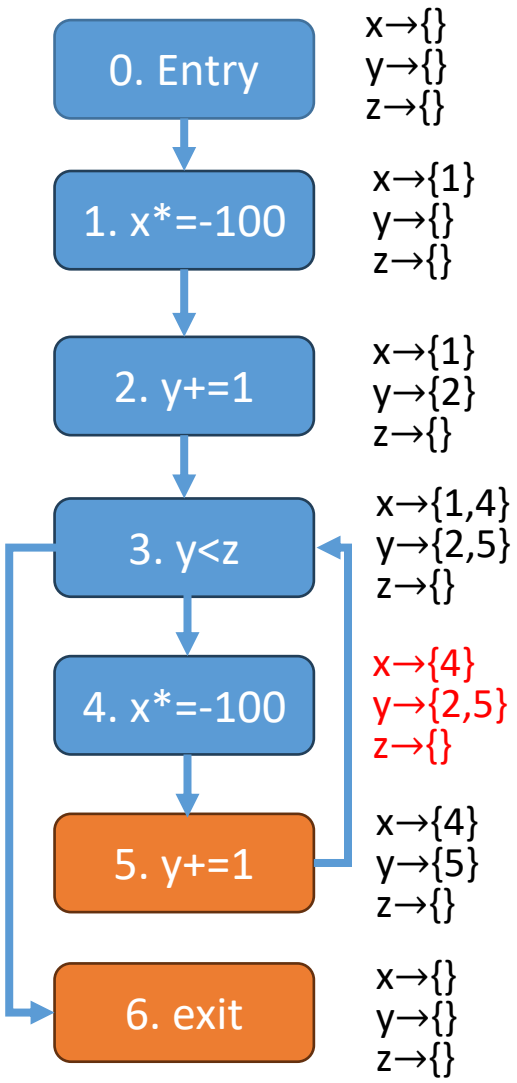
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{1}
y→{2}
z→{}
4. x*=-100
x→{4}
y→{2}
z→{}
5. y+=1
x→{4}
y→{5}
z→{}
6. exit
x→{}
y→{}
z→{}

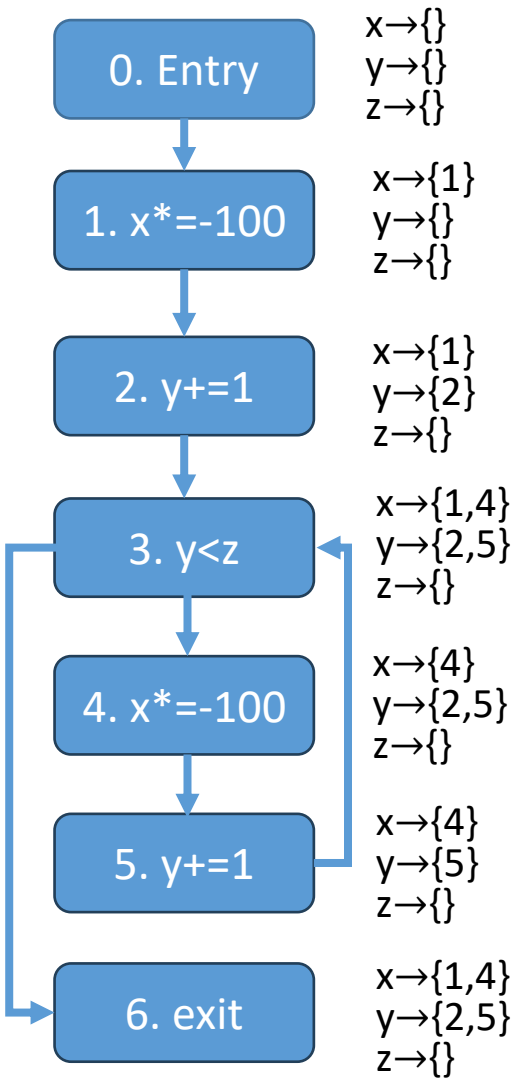
可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{1,4}
y→{2,5}
z→{}
4. x*=-100
x→{4}
y→{2}
z→{}
5. y+=1
x→{4}
y→{5}
z→{}
6. exit
x→{}
y→{}
z→{}

可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{1,4}
y→{2,5}
z→{}
4. x*=-100
x→{4}
y→{2,5}
z→{}
5. y+=1
x→{4}
y→{5}
z→{}
6. exit
x→{}
y→{}
z→{}

可达定值分析


0. Entry
x→{}
y→{}
z→{}
1. x*=-100
x→{1}
y→{}
z→{}
2. y+=1
x→{1}
y→{2}
z→{}
3. y<z
x→{1,4}
y→{2,5}
z→{}
4. x*=-100
x→{4}
y→{2,5}
z→{}
5. y+=1
x→{4}
y→{5}
z→{}
6. exit
x→{1,4}
y→{2,5}
z→{}

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. a=c+(b+10);
2. if (a>b)
3.   c = a+10;
4. return a;
```

1运行结束的时候可用表达式是b+10、c+(b+10)
2运行结束的时候可用表达式是b+10、c+(b+10)
3运行结束的时候可用表达式是b+10、a+10
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

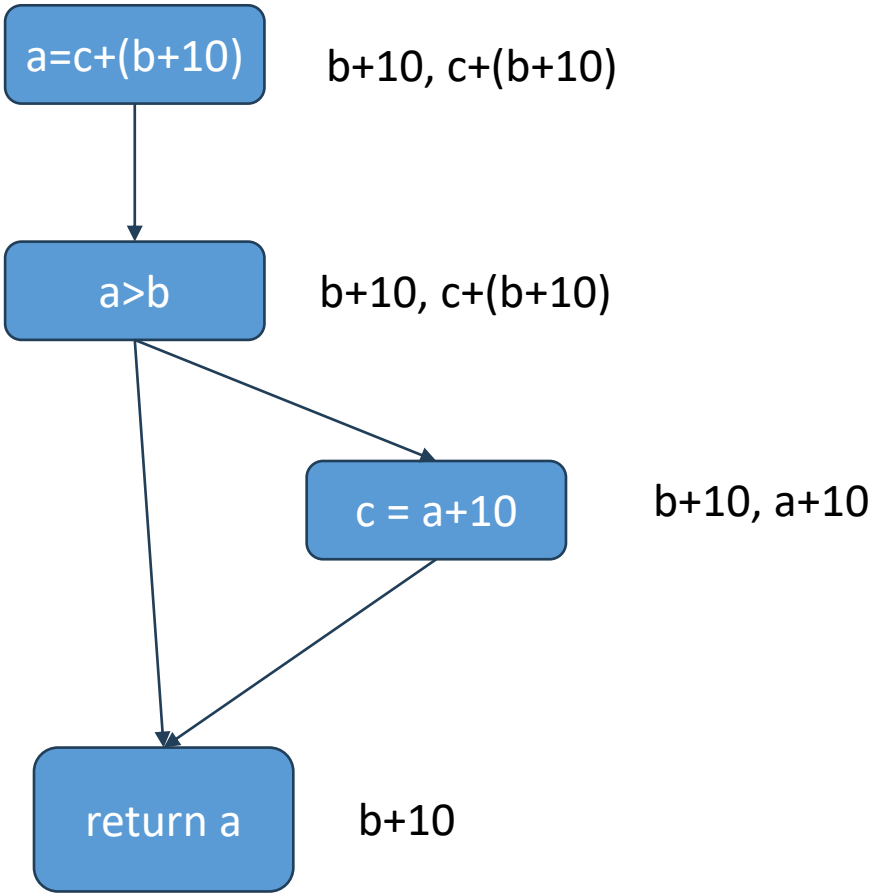
可用表达式分析——合并

- 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 ⊔ 乙)中

可用表达式分析-节点转换函数

- $f_v(\text{甲}) = (\text{甲} - \text{KILL}_\text{v}) \cup \text{GEN}_\text{v}$
  - 对于赋值语句x=...
    - KILL_v ={所有包含x的表达式}
    - GEN_v ={当前语句中求值的不含x的表达式}
  - 对于其他语句
    - KILL_v={}
    - GEN_v={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

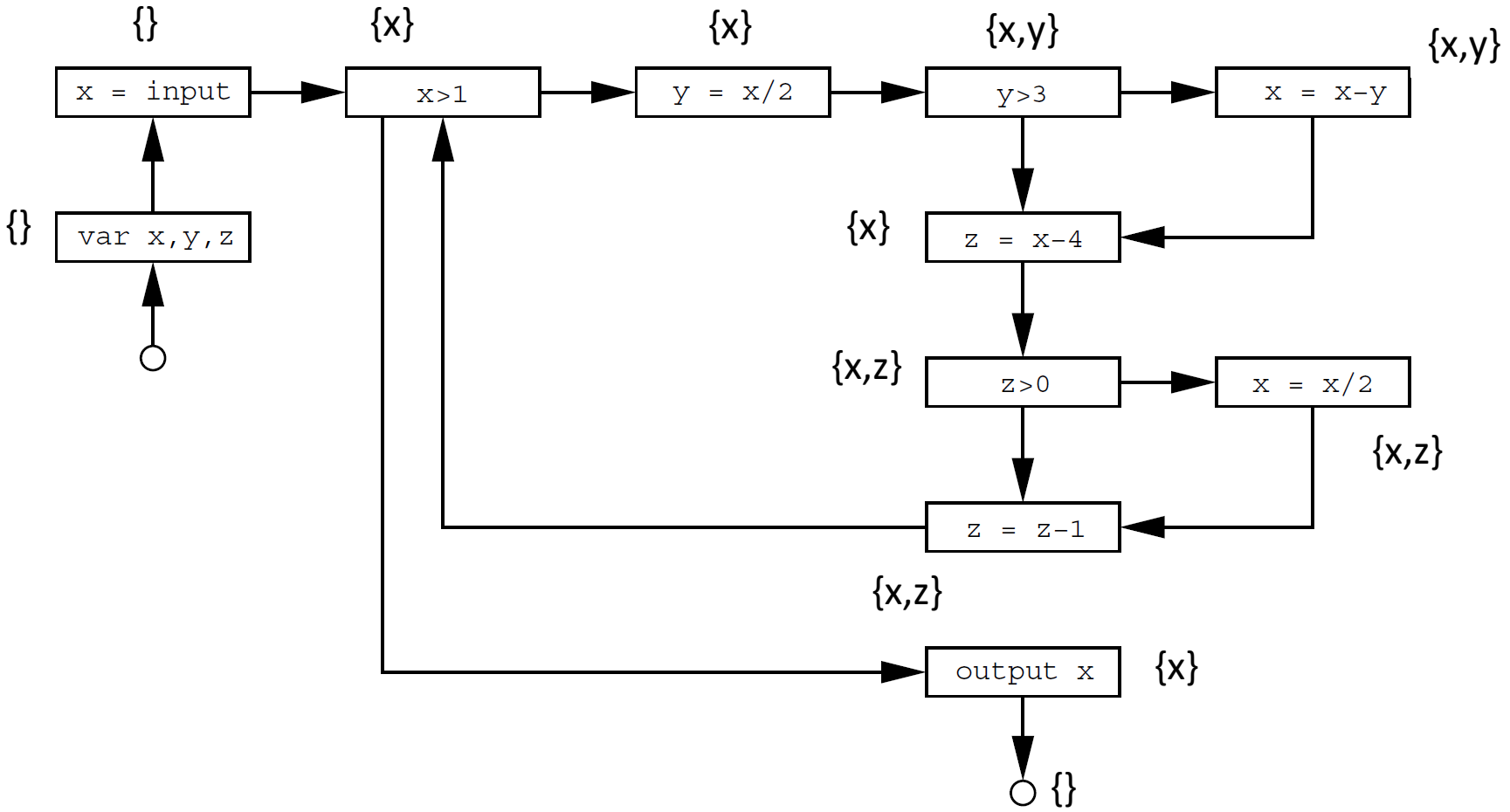
- 和之前的分析不同
  - 分析是下近似，即抽象域包含的为具体域的子集
  - 初始值为全集
  - 合并操作为求交
- 标准数据流框架同时提供对上近似和下近似的支持

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


{}
x = input
{x}
x>1
{x}
y = x/2
{x,y}
y>3
{x,y}
x = x-y
{}
var x,y,z
{x}
z = x-4
{x,z}
z>0
x = x/2
{x,z}
z = z-1
{x,z}
output x
{x}
{}

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 ⊔ 乙)中

活跃变量分析-节点转换函数

- $f_v(\text{甲}) = (\text{甲} \backslash \text{KILL}_v) \cup GEN_v$
  - GEN_v = v中读取的所有变量
  - $KILL_v = \begin{cases} \{x\} & v := \text{x} = \text{exp}; \\ \{x\} & v := \text{int x}; \\ \{\} & otherwise \end{cases}$
- 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

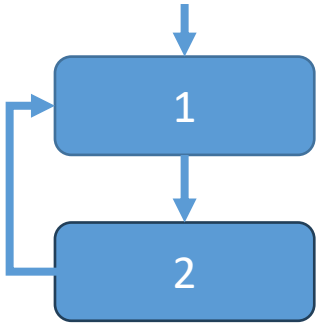
$IN_{exit} = I$
$\forall v \in (V - exit): IN_v \leftarrow \top$
$ToVisit \leftarrow V - exit$
$While(ToVisit.size > 0)\ \{$
  $v \leftarrow$ ToVisit中任意节点
  $ToVisit\ \text{-}= v$
  $OUT_v \leftarrow \sqcup_{w \in succ(v)} IN_w$
  $If\left(IN_v \neq f_v(OUT_v)\right)\ ToVisit \cup= pred(v)$
  $IN_v \leftarrow f_v(OUT_v)$
$\}$

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

- 给定任意终止的具体执行序列，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Exit开始反向对长度做归纳
- 问题：执行序列不一定在终止，不终止的怎么办?
  - 数据流分析只保证从entry开始（正向）或到exit结束（反向）的正确性，其他情况无法保证
  - 对活跃变量分析，可以单独引入证明
    - 但对于其他一些分析不一定能证明

活跃变量分析-正确性

- 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

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
    - 数据流框架只在结束的序列上讨论正确性

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，在所有终止执行中一定会被读取的表达式
  - 如果从某个程序点开始的所有执行都不终止，则可返回任意结果
- 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
- 请设计繁忙表达式分析。请给出分析方向、抽象域设计（抽象值集合、γ、初值）、转换函数、合并操作，并简要讨论正确性。

参考资料

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