

Lecture 6

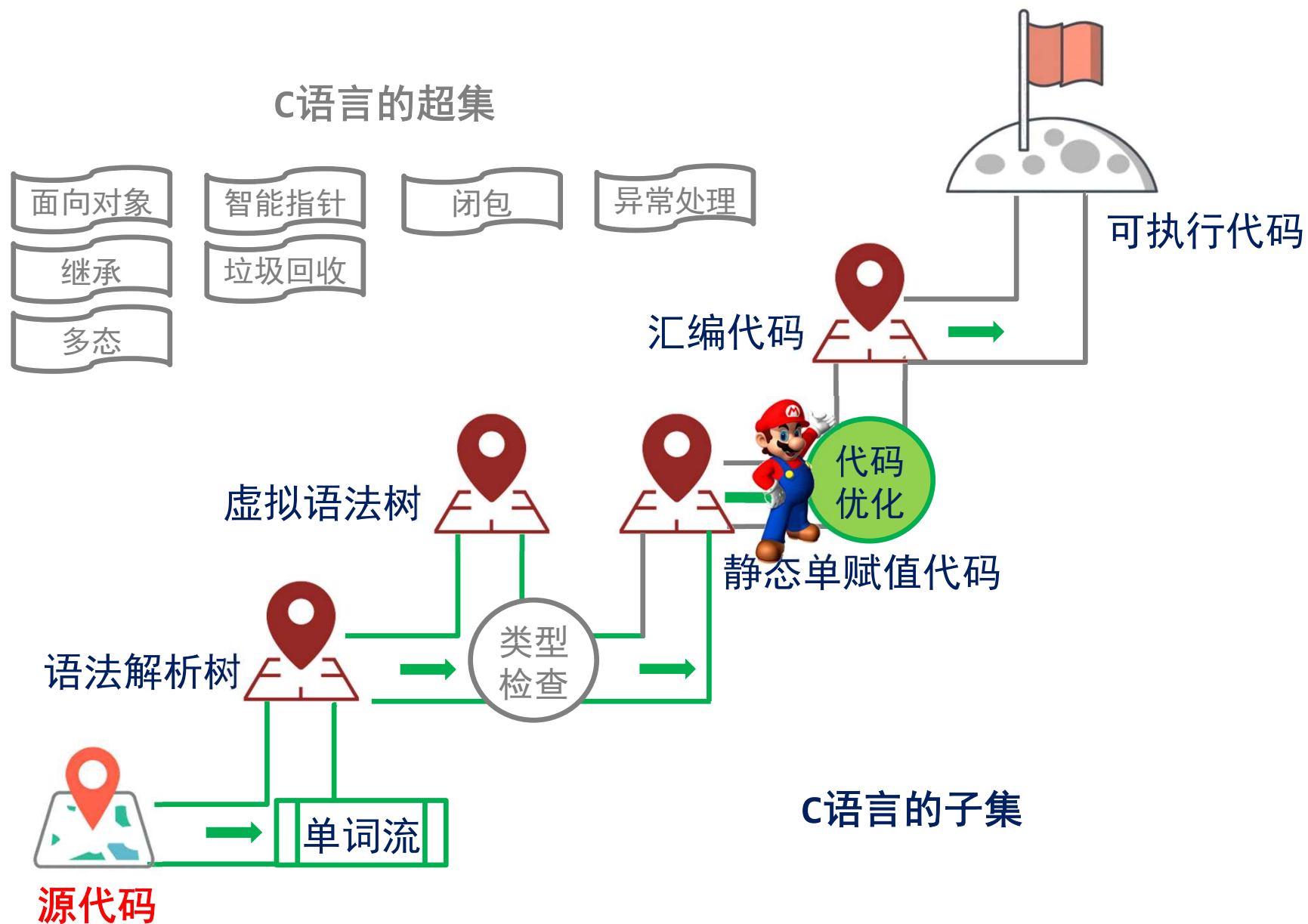
代码优化和程序分析

徐 辉

xuh@fudan.edu.cn



学习地图



大纲

- 一、代码优化问题
- 二、程序分析理论基础
- 三、数据流分析
- 四、指针分析

一、代码优化问题

LLVM的优化功能

```
#:opt -help
General options:

--O0      - Optimization level 0. Similar to clang -O0
--O1      - Optimization level 1. Similar to clang -O1
--O2      - Optimization level 2. Similar to clang -O2
--O3      - Optimization level 3. Similar to clang -O3
--Os      - Like -O2 with extra optimizations for size. Similar to clang -Os
--Oz      - Like -Os but reduces code size further. Similar to clang -Oz
...
```

```
#:llvm-as < /dev/null | opt -O1 -disable-output -debug-pass=Arguments
Pass Arguments: -tti -tbaa -scoped-noalias -assumption-cache-tracker -
targetlibinfo -verify -ee-instrument -simplifycfg -domtree -sroa -early-cse -
lower-expect
...
```

LLVM的优化passes

LLVM's Analysis and Transform Passes

- Introduction

- Analysis Passes

- -aa-eval: Exhaustive Alias Analysis Precision Evaluator
- -basic-aa: Basic Alias Analysis (stateless AA impl)
- -basiccg: Basic CallGraph Construction
- -count-aa: Count Alias Analysis Query Responses
- -da: Dependence Analysis
- -debug-aa: AA use debugger
- -domfrontier: Dominance Frontier Construction

- Transform Passes

- -adce: Aggressive Dead Code Elimination
- -always-inline: Inliner for `always_inline` functions
- -argpromotion: Promote 'by reference' arguments to scalars
- -bb-vectorize: Basic-Block Vectorization
- -block-placement: Profile Guided Basic Block Placement
- -break-crit-edges: Break critical edges in CFG
- -codegenprepare: Optimize for code generation
- -constmerge: Merge Duplicate Global Constants
- -dce: Dead Code Elimination

主要的优化点

- 函数优化
- 全局变量优化
- 无效代码优化
- 循环优化

函数优化

- 传参优化：
 - -argpromotion: argument promotion
 - 将函数参数的引用传递改为值传第
 - -deadargelim: dead argument elimination
 - 删除内部函数的无效参数
- 调用优化：
 - -inline: function inlining
 - -tailcallelim: tail call elimination
 - 将尾递归内联展开

全局变量优化

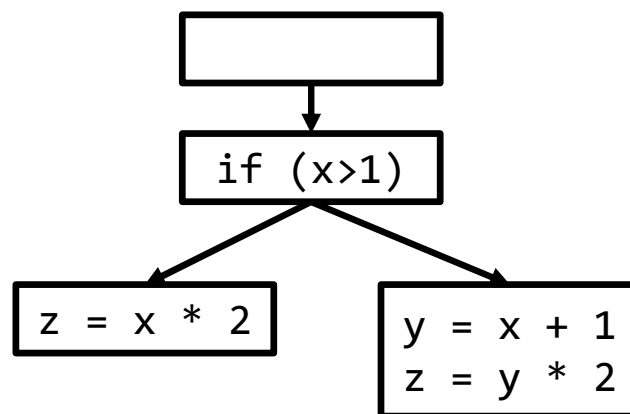
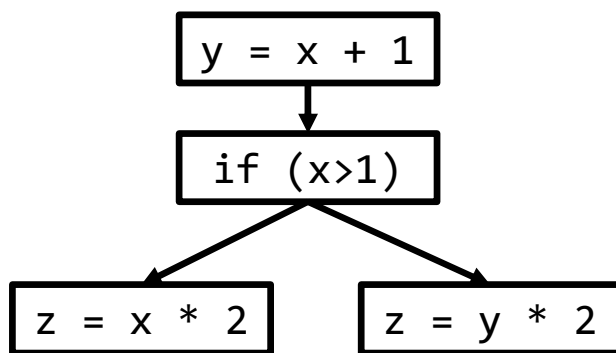
- `-constmerge`: Merge Duplicate Global Constants
 - 合并相同的全局常量数据，如字符串
- `-globaldce`: Dead Global Elimination
 - 分析使用到的全局变量，删除其余的
- `-globalopt`: Global Variable Optimizer
 - 将符合条件的全局变量替换为常量

常量优化

- -sccp: sparse conditional constant propagation
 - 分析变量是否为常量，如果是则替换为常量
- -ipsccp: interprocedural sparse conditional constant propagation

无效代码优化

- -dce: dead code elimination
 - 删除不可达代码;
- -deadtypeelim: dead type elimination
 - 删除无用类型定义
- -sink: code sinking
 - 将代码向后移动, 避免无用计算



循环优化

- -loop-unroll: unroll loops
 - 循环展开
- -licm: Loop invariant code motion
 - 将循环内的代码移到循环外，避免重复计算
- -loop-unswitch: unswitch loops
 - 将循环内的条件语句移到循环外

```
for (...)
    A
    if (lic)
        B
    C

if (lic)
    for (...)
        A; B; C
else
    for (...)
        A; C
```

代码优化的核心问题和技术

- 核心问题：如何实现代码等价变换，提升代码效率。
- 核心技术：程序分析
 - 数据流分析
 - 控制流分析

二、程序分析理论基础

程序分析难题

- 一般来讲，程序属性（program properties）是不可计算的（undecidability）
 - 如：一个变量的值是否为常量？
- 一个算法可以做到即可靠（sound）又完备（complete）吗？
 - 不能，或者不能保证该算法一定可以退出（terminate）
- 如何设计程序分析算法是一门艺术。

莱斯定理: Rice's Theorem

CLASSES OF RECURSIVELY ENUMERABLE SETS
AND THEIR DECISION PROBLEMS⁽¹⁾

BY
H. G. RICE

*“Any nontrivial property about the
Language recognized by a Turing
Machine is undecidable.”*

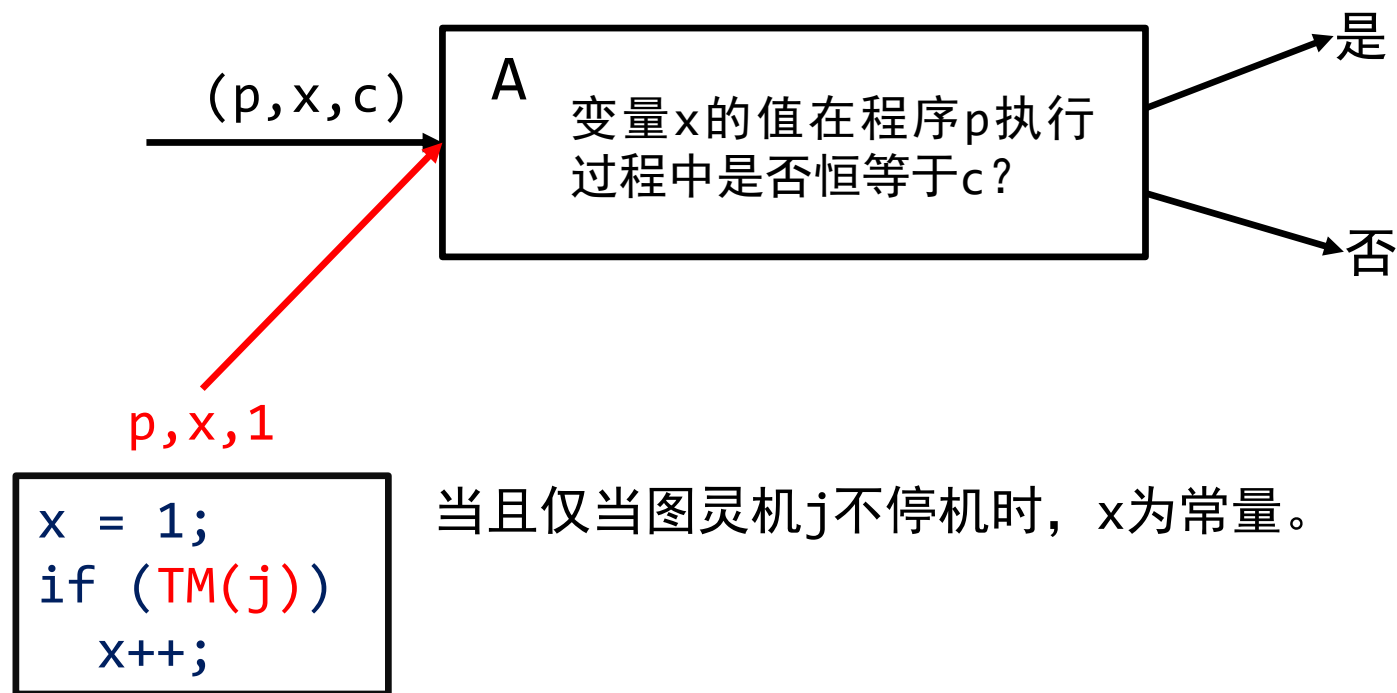
- Henry Gordon Rice, 1953

Revisit Chomsky Hierarchy

Class	Languages	Automaton	Rules	Word Problem	Example
type-0	recursively enumerable	Turing machine	no restriction	undecidable	Post's corresp. problem
type-1	context sensitive	linear-bounded TM	$\alpha \rightarrow \gamma$ $ \alpha \leq \gamma $	PSPACE-complete	$a^n b^n c^n$
type-2	context free	pushdown automaton	$A \rightarrow \gamma$	cubic	$a^n b^n$
type-3	regular	NFA / DFA	$A \rightarrow a$ or $A \rightarrow aB$	linear time	$a^* b^*$

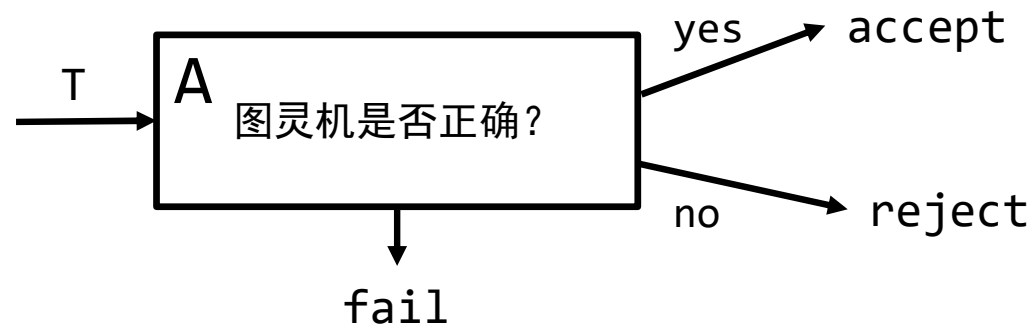
证明方法1：规约到图灵机停机问题

- 假设算法A可以分析任意程序p中的某个变量x的值是否为常量，
- 设计一个程序p证明算法A不存在？



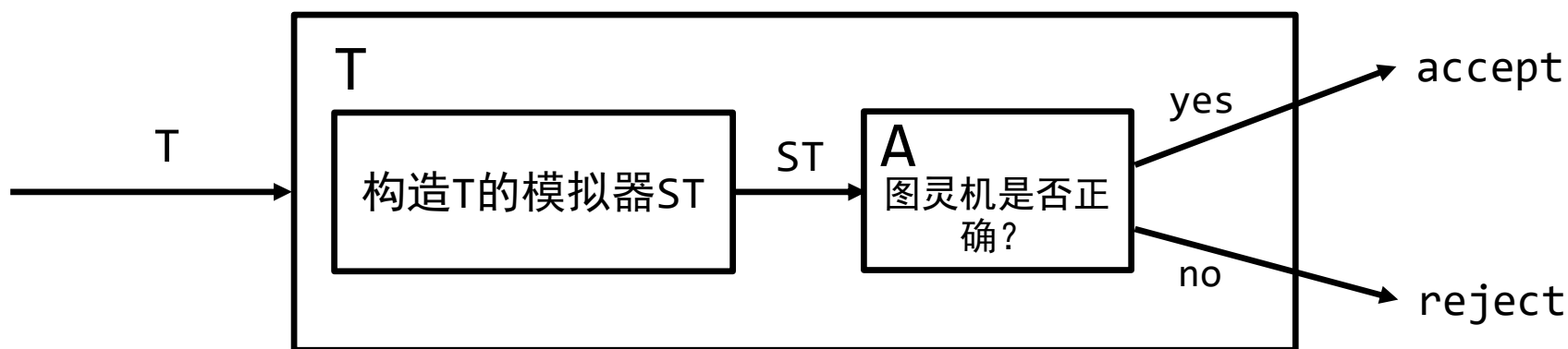
证明方法2: By Contradiction

- 假设算法A可以分析确定型图灵机T是否正确，输出accept或reject，如果无法分析则认为是fail。
 - 假设图灵机T正确指的是进入accept或reject状态；
 - 图灵机T错误是进入了fail状态。
- 如何构造矛盾？
 - 构造T的模拟器ST ($< |w| \text{ steps}$)
 - 如果T结束在accept，则ST进入fail；
 - 如果T结束在其它状态或未结束，则ST为accept或reject



证明方法2: By Contradiction

- 如果T的结果为accept, 则ST没有结束在fail状态; 根据构造方法, ST的模拟对象S此时不能为accept, 矛盾。
- 如果T的结果为reject, 则说明ST结束在fail状态; 根据构造方法, ST的模拟对象T此时应为accept, 矛盾。



三、数据流分析

数据流分析

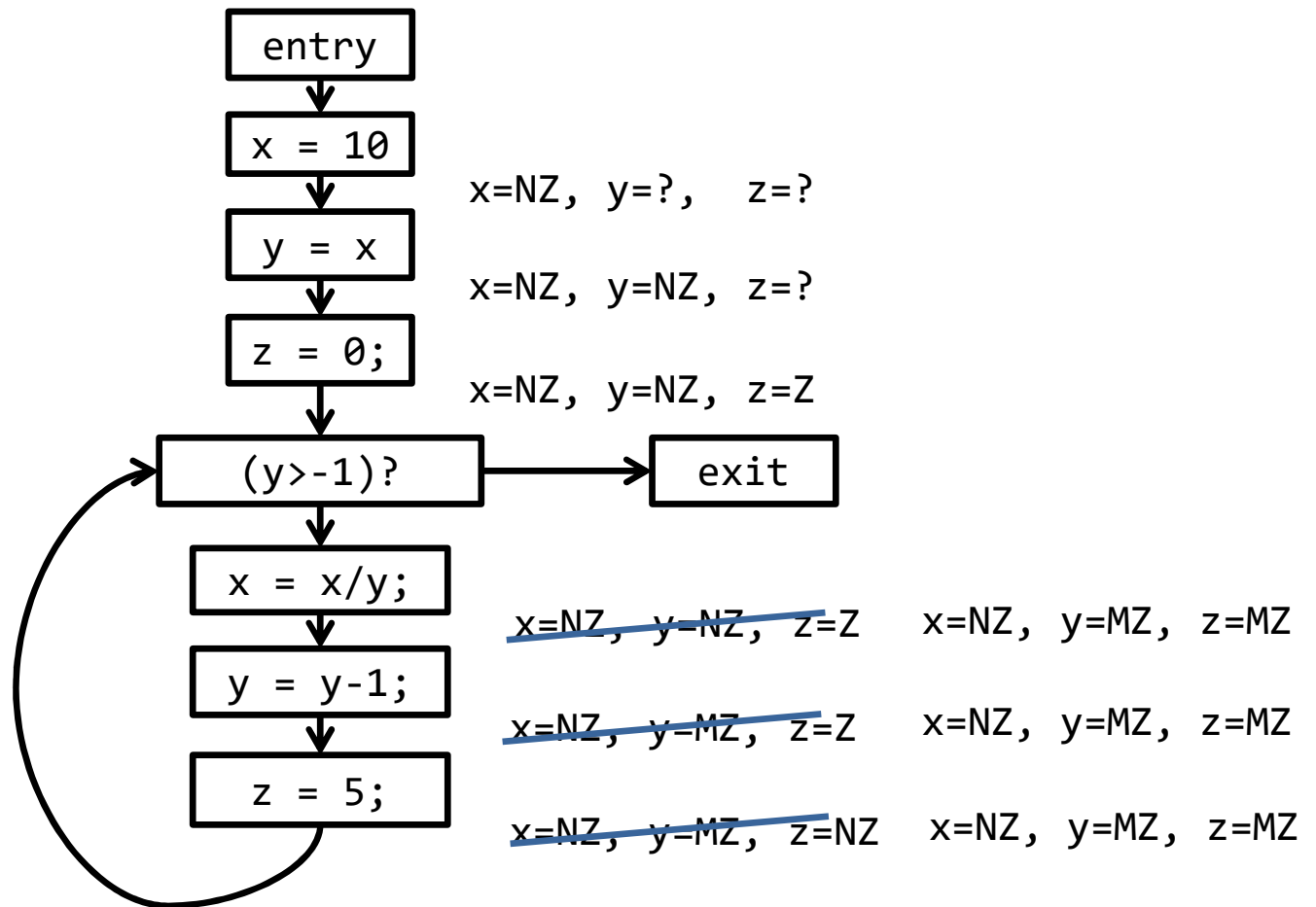
- 数据流分析：跟踪程序中的全部或一组变量的抽象值（abstract values）。
- 数据流分析问题举例：
 - 零值分析：变量x的取值可能为0吗？
 - 抽象值：{NZ, Z, MZ}
 - NZ: non-zero
 - Z: zero
 - MZ: maybe zero
 - 空指针分析：指针x可能为空指针吗？

如何分析除数为0的问题？

- 定义转移函数 (transfer function)
 - 有哪些运算符会影响变量取值？
 - $+$, $-$, $*$, $/$, $\%$...
- 定义合并函数 (Join function)
 - $\text{Join}(Z, Z) \Rightarrow Z$
 - $\text{Join}(NZ, NZ) \Rightarrow NZ$
 - $\text{Join}(Z, NZ) \Rightarrow MZ$
 - $\text{Join}(MZ, *) \Rightarrow MZ$

应用：分析除数 y 是否可能为0？

```
x = 10;  
y = x;  
z = 0;  
while (y > -1){  
    x = x/y;  
    y = y-1;  
    z = 5;  
}
```

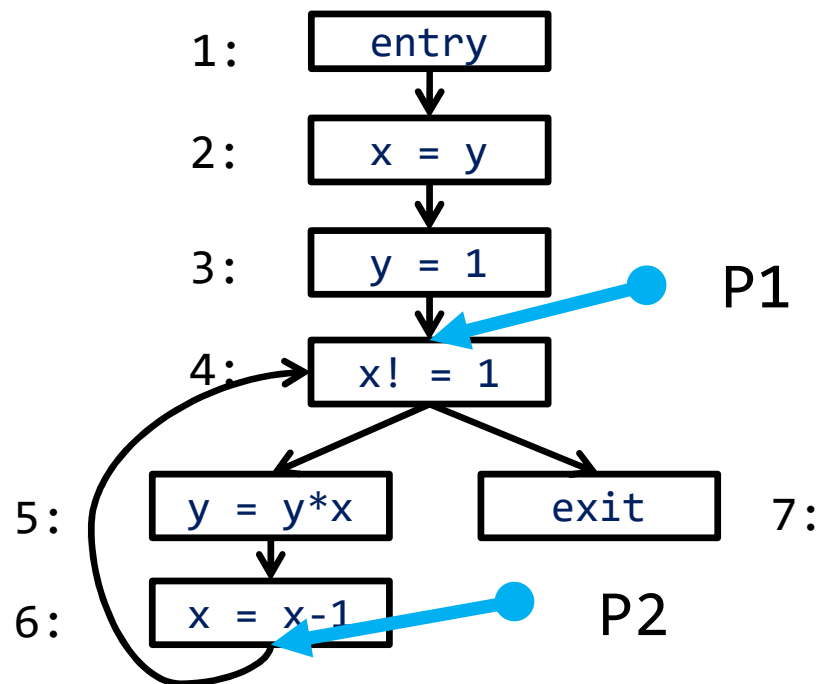


数据流分析问题

- 可达性分析: Reaching definition analysis
 - 可用于常量分析、查找未初始化的变量
- 繁忙表达式分析: Very busy expression analysis
 - 优化代码体积
- 可用表达式分析: Available expression analysis
 - 避免重复计算相同的表达式
- 活跃变量分析: Live variable analysis
 - 寄存器分配

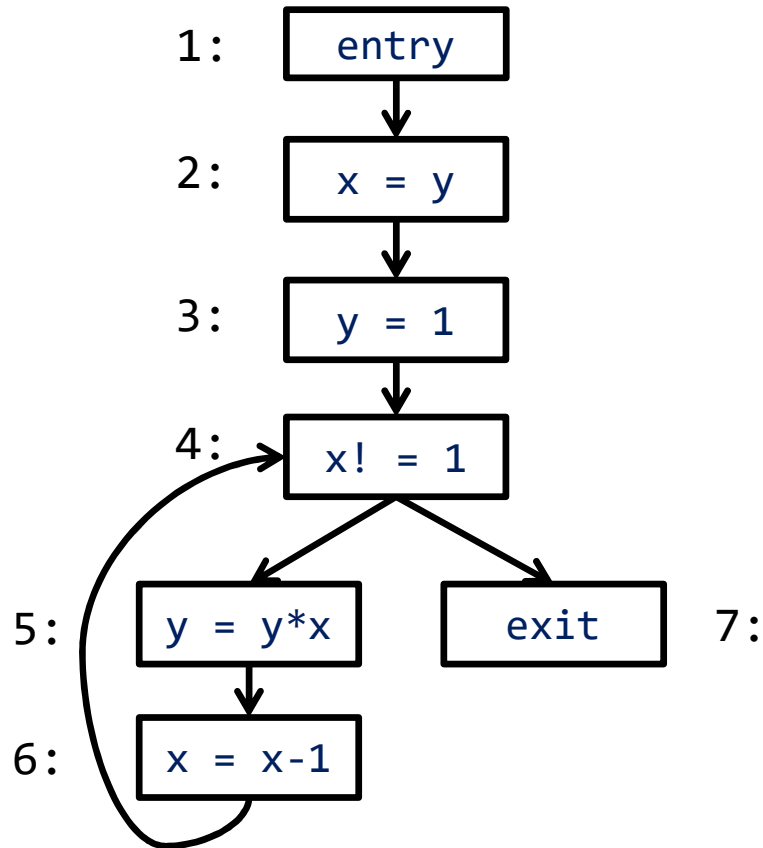
可达性分析

- 为每一个程序点（program point）分析某一变量的某个定义/赋值语句是否可达。



- $y = 1$ 可到达P1
- $y = 1$ 不能达到P2

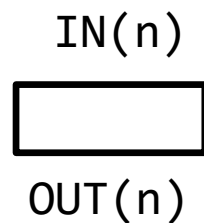
分析方法



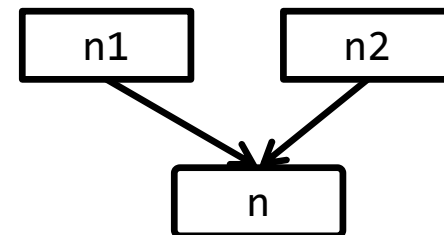
- 为每个节点分配一个编号
- 用 $IN(n)$ 表示节点的入向属性集合。
- 用 $OUT(n)$ 表示节点的出向属性集合。
- 遍历控制流图并应用Transfer和Join函数计算每一个节点的 $IN(n)$ 和 $OUT(n)$ 。
- 直到 $IN(n)$ 和 $OUT(n)$ 不再变化。

定义Transfer和Join函数

Transfer



Join



$$\text{OUT}(n) = (\text{IN}(n) - \text{KILL}(n)) \cup \text{Gen}(n)$$

$$\text{IN}(n) = \text{OUT}(n1) \cup \text{OUT}(n2)$$

n : cond=? Gen(n) = \emptyset
KILL(n) = \emptyset

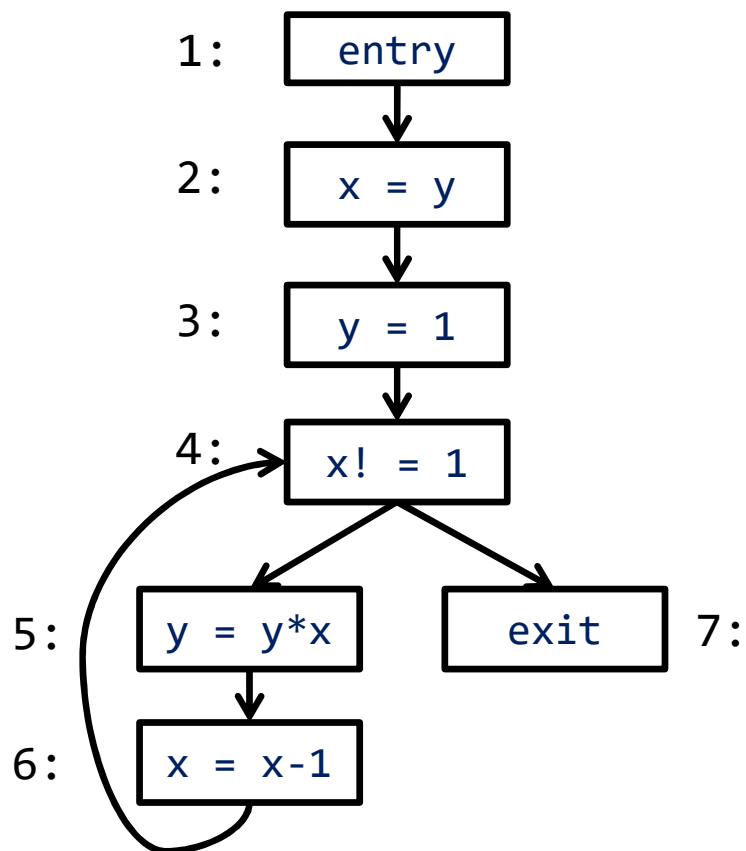
$$\text{IN}(n) = \bigcup_{n' \in \text{predecessor}(n)} \text{OUT}(n')$$

$$\begin{array}{l} n: \boxed{x=a} \quad \text{Gen}(n) = \{ \langle x, n \rangle \} \\ \quad \quad \quad \text{KILL}(n) = \{ \langle x, m \rangle : m \neq n \} \end{array}$$

Chaotic Iteration

```
For (each node n):  
    IN[n] = OUT[n] =  $\emptyset$   
OUT[entry] = {<v, ?>: v is a program variable}  
Repeat:  
    For(each node n):  
        For(each n's predecessor p)  
            IN[n] = IN[n]  $\cup$  OUT[p]  
            OUT(n)=(IN[n]-KILL(n))  $\cup$  Gen(n)  
Until IN[n] and OUT[n] stops changing for all n
```

应用举例



$\langle x, n \rangle$: 表示变量x在第n个节点被赋值

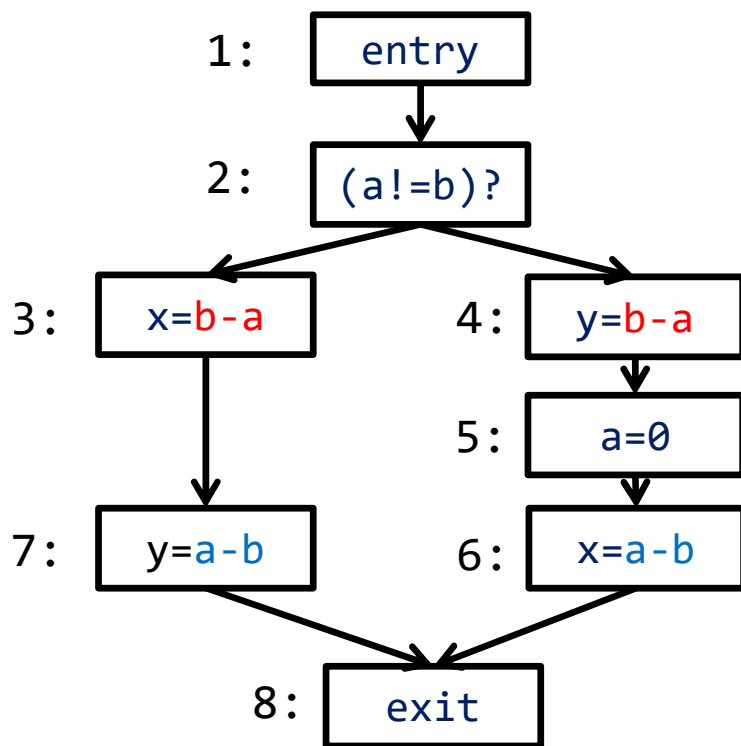
n	IN(n)	OUT(n)
1	-	$\{\langle x, ? \rangle \langle y, ? \rangle\}$
2	$\{\langle x, ? \rangle \langle y, ? \rangle\}$	$\{\langle x, 2 \rangle \langle y, ? \rangle\}$
3	$\{\langle x, 2 \rangle \langle y, ? \rangle\}$	$\{\langle x, 2 \rangle \langle y, 3 \rangle\}$
4	$\{\langle x, 2 \rangle \langle y, 3 \rangle \langle x, 6 \rangle \langle y, 5 \rangle\}$	$\{\langle x, 2 \rangle \langle y, 3 \rangle \langle x, 6 \rangle \langle y, 5 \rangle\}$
5	$\{\langle x, 2 \rangle \langle y, 3 \rangle \langle x, 6 \rangle \langle y, 5 \rangle\}$	$\{\langle x, 2 \rangle \langle y, 5 \rangle \langle x, 6 \rangle\}$
6	$\{\langle x, 2 \rangle \langle y, 5 \rangle \langle x, 6 \rangle\}$	$\{\langle x, 6 \rangle \langle y, 5 \rangle\}$
7	$\{\langle x, 2 \rangle \langle y, 3 \rangle \langle x, 6 \rangle \langle y, 5 \rangle\}$	$\{\langle x, 2 \rangle \langle y, 3 \rangle \langle x, 6 \rangle \langle y, 5 \rangle\}$

算法是否一定会终止？

- 可达定义分析的迭代算法一定会终止
 - Join和Transfer的两个函数是单调的 (monotonic)
 - IN和OUT集合元素数目只会增加，不会减少。
 - IN集合OUT不可能无限扩大，最大是程序中所有定义语句的集合。
 - IN和OUT一定会在某一轮迭代后停止改变。

繁忙表达式分析

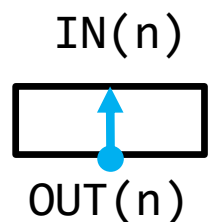
- 繁忙表达式：多条路径中都存在的公共表达式，且无论走那条路径，该表达式在其操作数被重新赋值之前被执行。



- $b-a$ 是繁忙表达式；
- $a-b$ 不是繁忙表达式

定义Transfer和Meet函数

Transfer

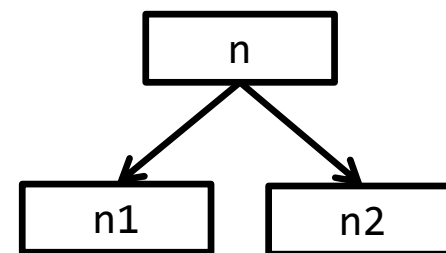


$$\text{IN}(n) = (\text{OUT}(n) - \text{KILL}(n)) \cup \text{Gen}(n)$$

[illegible]

n : $x=a$ $\text{Gen}(n) = \{a\}$
 $\text{KILL}(n) = \{\text{expr } e: e \text{ contains } x\}$

Meet



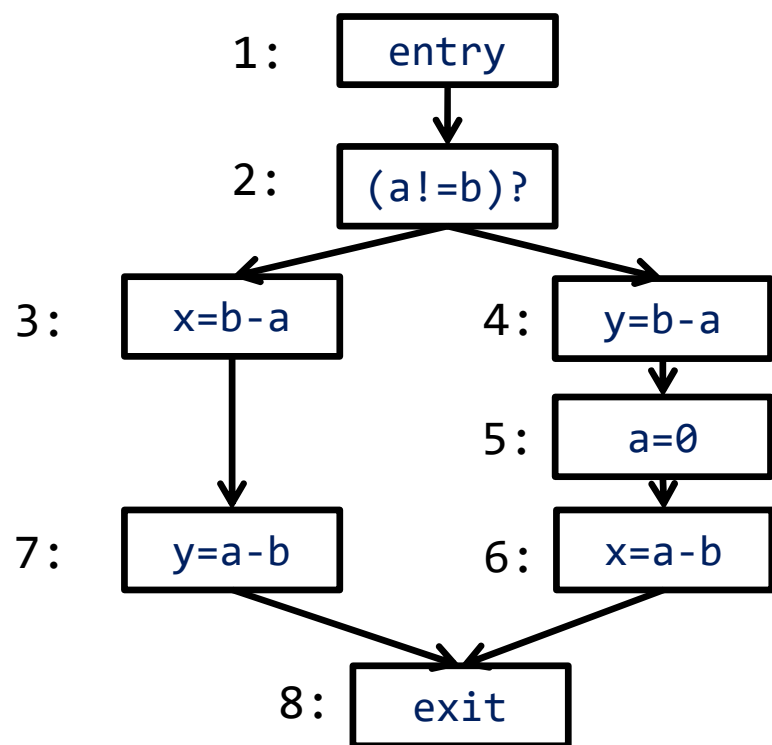
$$\text{OUT}(n) = \text{IN}(n1) \cap \text{IN}(n2)$$

$$\text{OUT}(n) = \bigcap_{n' \in \text{successor}(n)} \text{IN}(n')$$

Chaotic Iteration

```
For (each node n):  
    IN[n] = OUT[n] = set of all expressions in program  
IN[exit] =  $\emptyset$   
Repeat:  
    For(each node n):  
        For(each n's successor s):  
            OUT[n] = OUT[n]  $\cup$  IN[s]  
            IN[n]=(OUT[n]-KILL(n))  $\cup$  Gen(n)  
Until IN[n] and OUT[n] stops changing for all n
```

练习

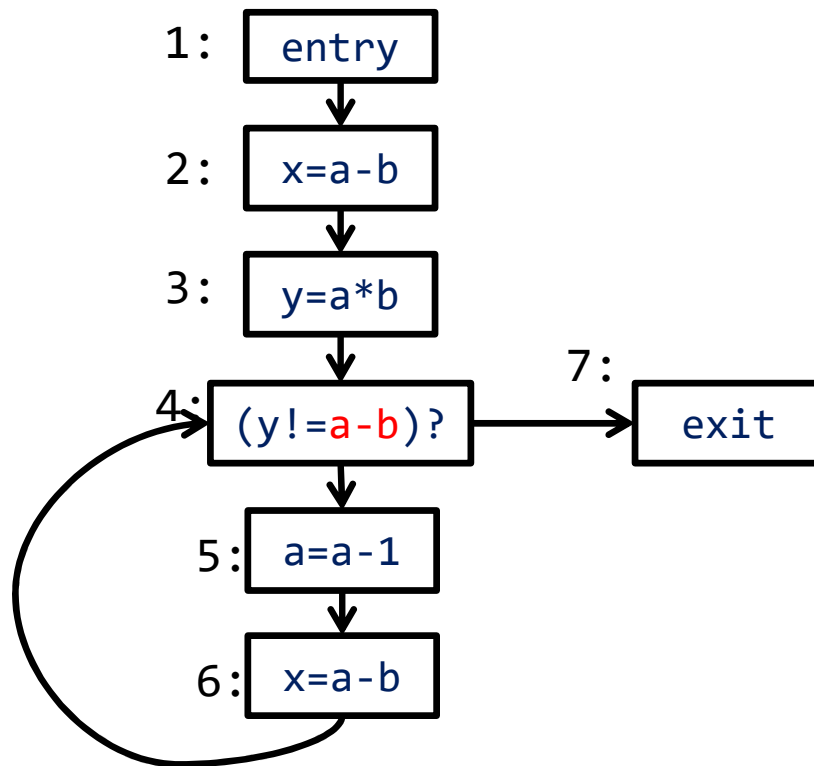


假设所有的节点都初始化为可用表达式集合：
 $\{a-b, b-a\}$

n	IN(n)	OUT(n)
1	-	$\{b-a\}$
2	$\{b-a\}$	$\{b-a\}$
3	$\{a-b, b-a\}$	$\{a-b\}$
4	$\{b-a\}$	\emptyset
5	\emptyset	$\{a-b\}$
6	$\{a-b\}$	\emptyset
7	$\{a-b\}$	\emptyset
8	\emptyset	-

可用表达式分析

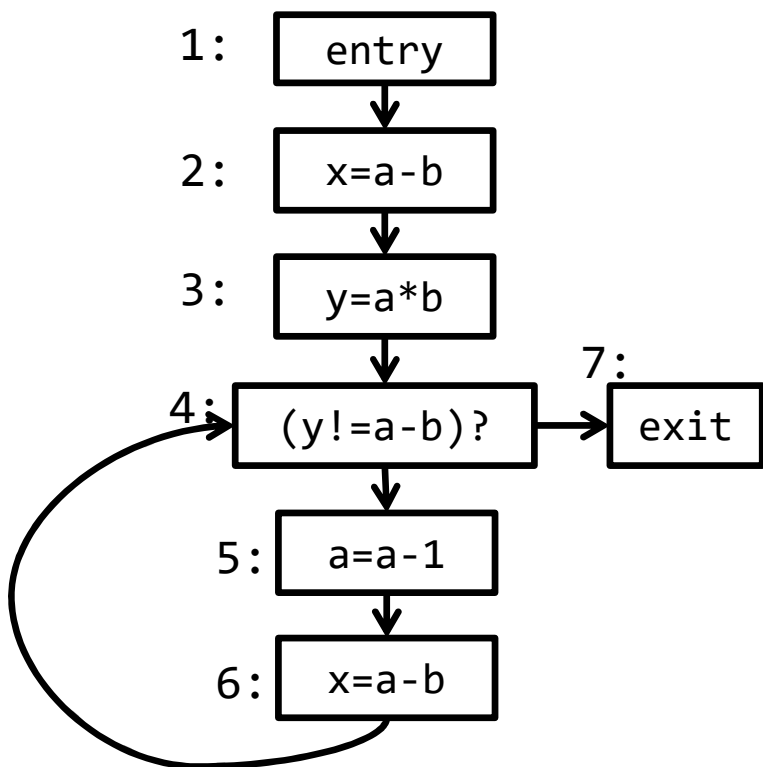
- 可用表达式：无论走那条路径，该表达式的操作数都未重新赋值，避免重复计算表达式的值。



- 避免重复计算 $a - b$ 的值

练习

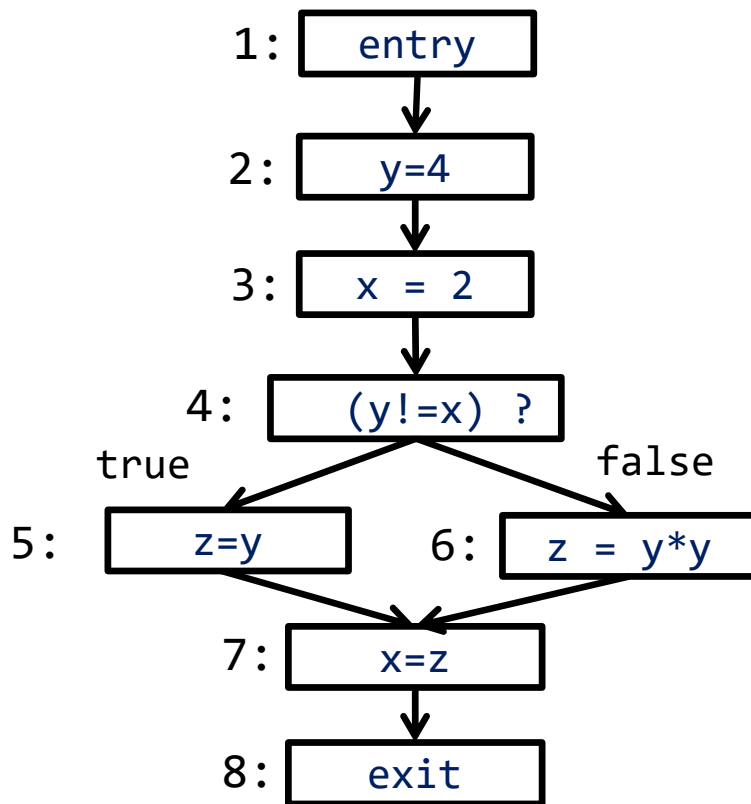
前向分析



n	IN[n]	OUT[n]
1	--	\emptyset
2	\emptyset	{a-b}
3	{a-b}	{a-b, a*b}
4	{a-b, a*b }	{a-b, a*b }
5	{a-b, a*b }	\emptyset
6	\emptyset	{a-b}
7	{a-b, a*b }	--

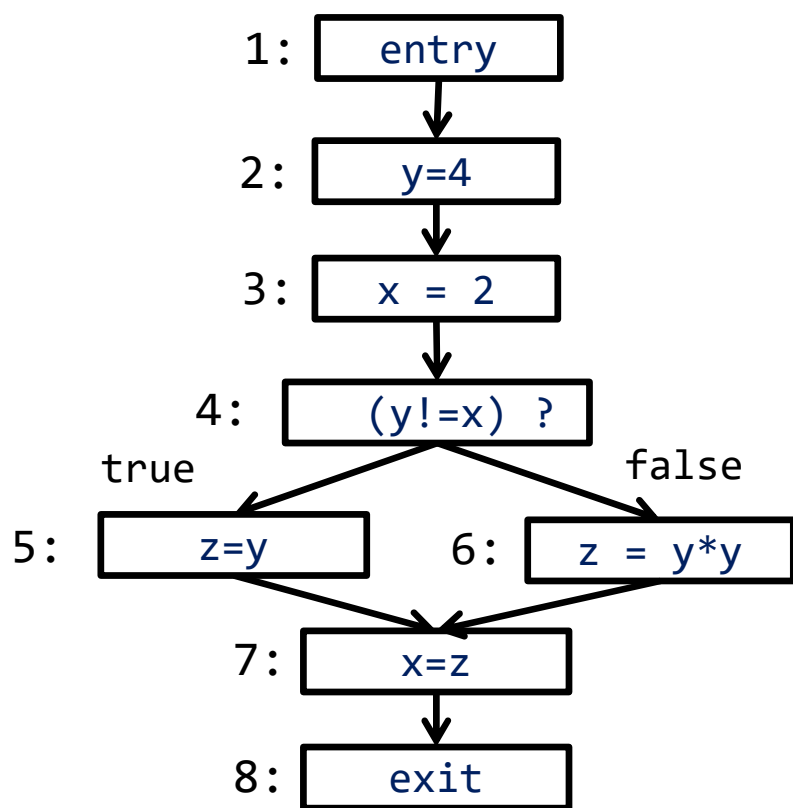
活跃变量分析

- 活跃变量：一个变量在被重新赋值前被使用(def-use)。



- x: 3-4-5-6
- y: 2-3-4-5-6
- z: 5-6-7

练习



后向分析

n	IN[n]	OUT[n]
1	--	\emptyset
2	\emptyset	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	\emptyset
8	\emptyset	--

数据流分析方法

$$\boxed{} [n] = (\boxed{} [n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{} [n] = \boxed{} \boxed{} [n']$$

$$n' \in \boxed{}(n)$$

$$\boxed{} = \text{IN or OUT} \quad \boxed{} = U \text{ (may) or } n \text{ (must)}$$

$$\boxed{} = \text{predecessors or successors}$$

可达性分析

$$\boxed{\text{OUT}}[n] = (\boxed{\text{IN}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{IN}}[n] = \boxed{\text{U}} \boxed{\text{OUT}}[n']$$

$$n' \in \boxed{\text{pred.}}(n)$$

$\boxed{}$	= IN or OUT	$\boxed{}$	= U (may) or n (must)
$\boxed{}$		$\boxed{}$	= predecessors or successors

繁忙表达式分析

$$\boxed{\text{IN}}[n] = (\boxed{\text{OUT}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{OUT}}[n] = \boxed{n} \boxed{\text{IN}}[n']$$

$$n' \in \boxed{\text{succ.}}(n)$$

$\boxed{}$	= IN or OUT	$\boxed{}$	= U (may) or n (must)
$\boxed{}$		$\boxed{}$	= predecessors or successors

可用表达式分析

$$\boxed{\text{OUT}}[n] = (\boxed{\text{IN}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{IN}}[n] = \boxed{n} \boxed{\text{OUT}}[n']$$

$$n' \in \boxed{\text{pred.}}(n)$$

$\boxed{}$	= IN or OUT	$\boxed{}$	= U (may) or n (must)
$\boxed{}$		$\boxed{}$	= predecessors or successors

活跃变量分析

$$\boxed{\text{IN}}[n] = (\boxed{\text{OUT}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{OUT}}[n] = \boxed{\text{U}} \boxed{\text{IN}}[n']$$

$$n' \in \boxed{\text{succ.}}(n)$$

$$\begin{array}{ll} \boxed{\phantom{\text{IN}}} = \text{IN or OUT} & \boxed{\phantom{\text{U}}} = \text{U (may) or n (must)} \\ \boxed{\phantom{\text{OUT}}} & \boxed{\phantom{\text{predecessors}}} = \text{predecessors or successors} \end{array}$$

数据流分析任务总结

	May Analysis (U)	Must Analysis (n)
前向分析	可达性分析	可用表达式分析
后向分析	活跃变量分析	繁忙表达式分析

- 可达定义分析
- 繁忙表达式分析
- 可用表达式分析
- 活跃变量分析

四、指针分析

指针分析问题

- 指针别名（Pointer alias）分析：判断两个指针是否在某一时刻指向同一内存地址。
- 指针指向（Points-to）分析：分析指针指向的内存地址，分析结果可用于指针别名分析。

```
Circle x = new Circle();  
x.radius = 1;  
y = x.radius;  
assert(y == 1)
```

```
Circle x = new Circle();  
Circle z = ?  
x.radius = 1;  
z.radius = 2;  
y = x.radius;  
assert(y == 1)
```

别名确定性

- May Alias
 - 如果两个指针是may alias, 那么它们可能存在别名关系;
 - 反之则一定不存在别名关系 (must-not alias) 。
- Must Alias:
 - 如果两个指针是must alias, 那么它们一定存在别名关系。
- 哪一个问题更容易? 哪一种分析更有用?

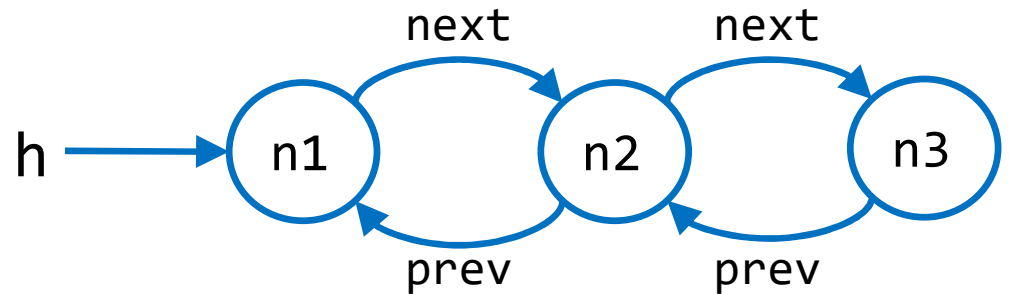
指针分析的用途

- 编译器优化（保守策略）
 - 常量传播（constant propagation）
 - 下列代码中x是常量吗？
 - 如果*p和x一定不是alias，则是；
 - 如果*p和x一定是alias，则否；
 - 如果*p和x可能是alias，则否。
 - 无效代码删除（dead code elimination）
 - ...

```
x = 3;  
*p = 4;
```

为什么指针分析很难

```
class Node {  
    int data;  
    Node next, prev;  
}  
  
Node h = null;  
for (...) {  
    Node v = new Node();  
    if (h != null) {  
        v.next = h;  
        h.prev = v;  
    }  
    h = v;  
}
```



h.data
h.next.prev.data
h.next.next.prev.data
h.next.prev.next.prev.data
...

需要近似分析

- 一般来讲，指针分析问题是undecidable的问题
 - 需在soundness、completeness和termination中取舍
- 通常选择保留soundness、牺牲completeness
 - 造成false positives
 - 不会有false negatives
 - 无论程序如何运行

```
Circle x = new Circle();  
Circle z = ?  
x.radius = 1;  
z.radius = 2;  
y = x.radius;  
assert(y == 1)
```

哪些操作可能造成Alias?

- 指针
- 函数调用传参（指针、引用）
- 数组索引

```
int *p, i;  
p = &i;
```

```
void foo(Object a, Object b) { ... }  
foo(x,x); // a和b在函数foo中是alias
```

```
int i,j,a[100];  
i = j; // a[i] and a[j] alias
```

指针分析建模方法

- 不同分析方法实际精度存在一定区别，主要体现在
 - Flow-sensitivity: 是否考虑代码执行顺序
 - Flow sensitive: 计算每一个程序点的指针指向;
 - Flow insensitive: 计算任意程序点可能的指向。
 - Path-sensitivity: 是否考虑控制流
 - Path sensitive: 分析过程只考虑单条特定控制流;
 - Path insensitive: 分析过程不区分控制流;
 - Context-sensitivity: 是否考虑函数调用
 - Context sensitive: 支持跨函数调用分析;
 - Context insensitive: 以函数为分析边界。
 - 如何对堆内存进行抽象
 - 以内存分配 (malloc、new) 为内存单元
 - 以每种类型为内存单元 (粗粒度)

指针分析算法

- Andersen-style Analyses
- Steensgaard-style Analyses

指针分析表示方法

- 别名对: alias pairs
 - 如 $*p$ 和 $*q$ 、 x 和 $*p$ 、 x 和 $*q$
- 等价集合: equivalence sets
 - 如 $\{*p, x, *q\}$
- 指针指向: point-to
 - $p \rightarrow x, q \rightarrow x$

```
int x;  
p = &x;  
q = p;
```

Andersen-style指针分析思路

- 将指针赋值视作子集约束
- 通过约束表示和传递指针指向信息
- Flow-insensitive
 - 不考虑语句顺序
- Context-insensitive
 - 与函数如何被调用无关
- 主要步骤
 - 1) 将指针指向关系映射为子集约束；
 - 2) 初始化约束图；
 - 3) 计算传递闭包更新约束关系。

提取约束关系

约束类型	赋值语句	约束	含义
Base	$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	$a = *b$	$a \supseteq^* b$	$\forall v \in pts(b), pts(a) \supseteq pts(v)$
Complex	$*a = b$	$*a \supseteq b$	$\forall v \in pts(a), pts(v) \supseteq pts(b)$

```
p = &a  
q = &b  
*p = q;  
r = &c;  
s = p;  
t = *p;  
*s = r;
```

```
p  $\supseteq$  {a}  
q  $\supseteq$  {b}  
*p  $\supseteq$  q  
r  $\supseteq$  {c}  
s  $\supseteq$  p  
t  $\supseteq$  *p  
*s  $\supseteq$  r
```

初始化约束图

- 约束图
 - 点表示变量的指针指向；
 - 边表示特定约束关系；
- 初始化
 - 包含关系：箭头
 - 指针指向：{ }

赋值语句	约束	含义	边
$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$	no edge
$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \rightarrow a$
$a = *b$	$a \supseteq^* b$	$\forall v \in pts(b), pts(a) \supseteq pts(v)$	no edge
$*a = b$	$*a \supseteq b$	$\forall v \in pts(a), pts(v) \supseteq pts(b)$	no edge

$p \supseteq \{a\}$

$q \supseteq \{b\}$

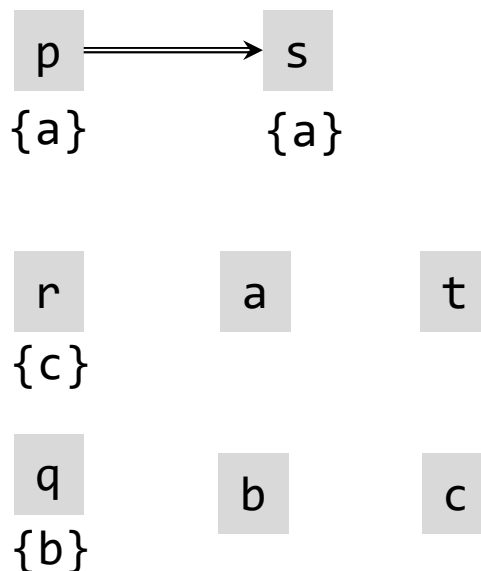
$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$



更新约束关系: Worklist算法

假设约束图已经初始化

Let $W = \{ v \mid \text{pts}(v) \neq \emptyset \}$ (所有指向集合非空的节点)

While W not empty

$v \leftarrow \text{select from } W$

 for each $a \in \text{pts}(v)$ do

 for each constraint $p \supseteq *v$

 add edge $a \rightarrow p$, and add a to W if edge is new

 for each constraint $*v \supseteq q$

 add edge $q \rightarrow a$, and add q to W if edge is new

 for each edge $v \rightarrow q$ do

$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(v)$, and add q to W if $\text{pts}(q)$ changed

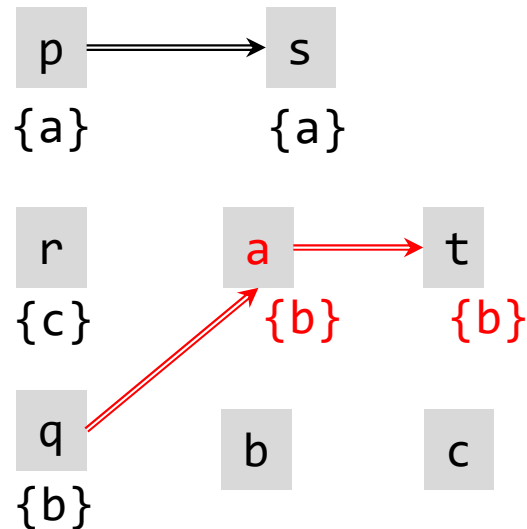
更新约束关系

```

p  $\supseteq$  {a}
q  $\supseteq$  {b}
*p  $\supseteq$  q
r  $\supseteq$  {c}
s  $\supseteq$  p
t  $\supseteq$  *p
*s  $\supseteq$  r

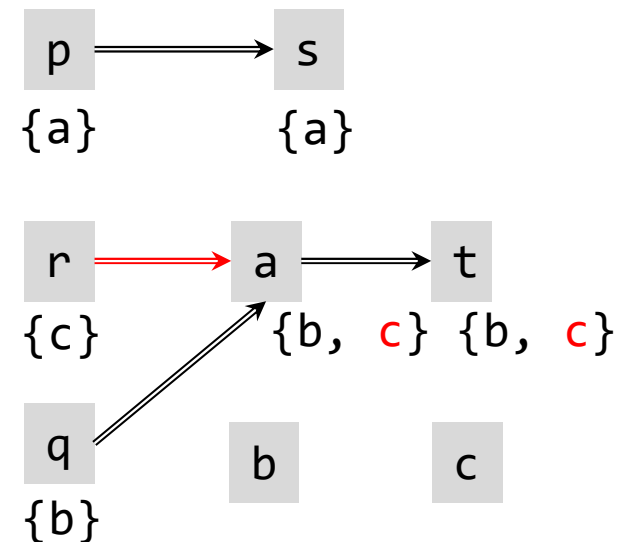
for each a  $\in$  pts(v) do
  for each constraint p  $\supseteq$  *v
    add edge a  $\rightarrow$  p, and add a to W if edge is new
  for each constraint *v  $\supseteq$  q
    add edge q  $\rightarrow$  a, and add q to W if edge is new
for each edge v  $\rightarrow$  q do
  pts(q) = pts(q)  $\cup$  pts(v), and add q to W if pts(q) changed
    
```

Step 1: Worklist: {**p**, s, r, q}



Result Worklist: {~~p~~, s, r, q, **a**, **t**}

Step 2: Worklist: {**s**, r, q, a, t}



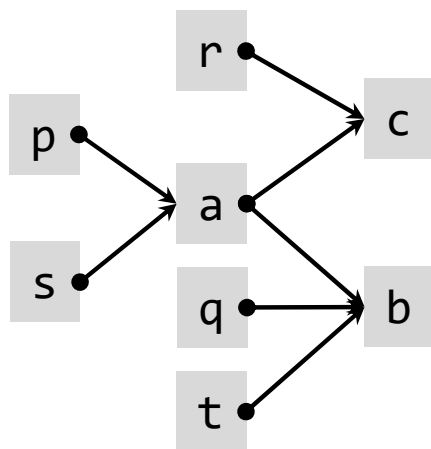
Result Worklist: {~~s~~, r, q, a, t}

精确性和算法复杂度

- 分析粒度较粗，相比flow sensitive存在误报。
 - *t和c不应为alias
- 复杂度 $O(n^3)$ ， n 是约束图的节点数。

```
p = &a  
q = &b  
*p = q;  
r = &c;  
s = p;  
t = *p;  
*s = r;
```

Flow Sensitive
分析结果



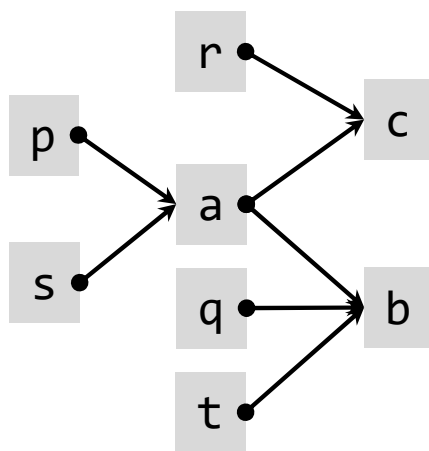
Andersen算法结果

```
pts(p) = {a}  
pts(q) = {b}  
pts(r) = {c}  
pts(s) = {a}  
pts(t) = {b, c}  
pts(a) = {b, c}  
pts(b) = ∅  
pts(c) = ∅
```

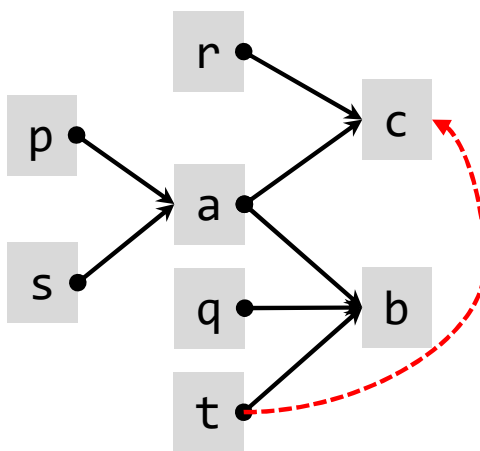
如何进一步优化性能？

- Andersen-style: 每个变量可以是任意变量的alias
 - $O(n^2)$ 空间
- 降低每个变量使用的空间（边数）？可以牺牲一些精度。
- 只有一条出边：将每个变量关联到一个abstract location
 - 如果 $*x$ 和 $*y$ 如果是alias, x 和 y 指向同一个abstract location
 - 接近线性复杂度 $O(n * \alpha(n))$

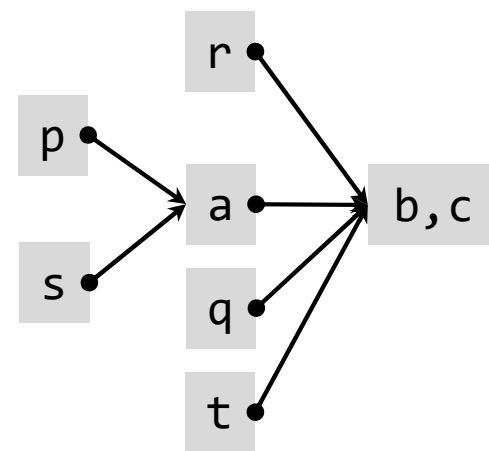
Flow Sensitive
分析结果



Andersen-style
分析结果



Steensgaard-style
分析结果



精度损失: $a = \&b$, $a = \&c$, b 和 c 不应是alias

Steensgaard-Style分析思路

- 使用等价约束 (equality constraints) 而非子集约束;
- 基于并查集的方法, 如果 $x=y$, 则 x 和 y 联通
- 接近线性复杂度 $O(n * \alpha(n))$, 粒度比Andersen-style更粗

约束类型	赋值语句	约束	含义	注释
Base	$a = \&b$	$a \subseteq \{b\}$	$loc(b) \subseteq pts(a)$	Steensgaard
		$a = \{b\}$	$loc(b) = pts(a)$	简化版 (便于理解)
Simple	$a = b$	$a = b$	$pts(a) = pts(b)$	
Complex	$a = *b$	$a = * b$	$\forall v \in pts(b), pts(a) = pts(v)$	
Complex	$*a = b$	$* a = b$	$\forall v \in pts(a), pts(v) = pts(b)$	

简化版的问题: $a = \&c, b = \&c$, a 和 b 不应是alias

并查集算法（简化版）

- 维护不存在相交关系的集合，支持查找和联合两种操作
 - Find(x): 返回包含变量x的集合
 - Union(x, y): 联合包含x和y的两个集合

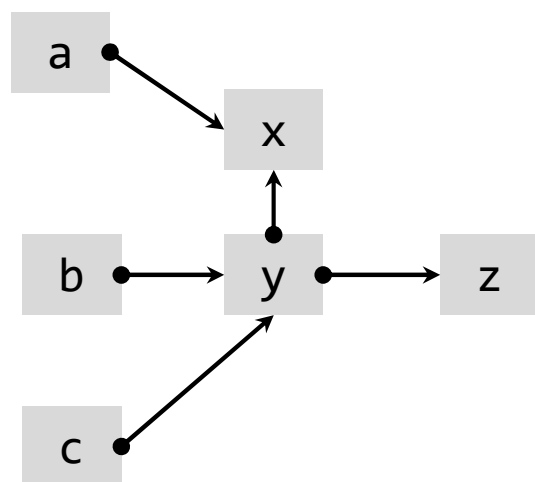
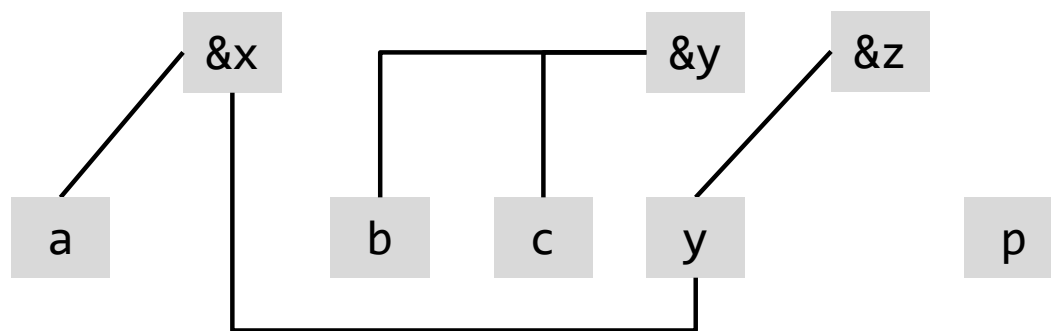
```
while(getPair()!=NULL){  
    [p,q] = readPair(p,q);  
    pset = find(p);  
    qset = find(q);  
    if(pset == qset)  
        continue;  
    else union(p,q);  
}
```


示例

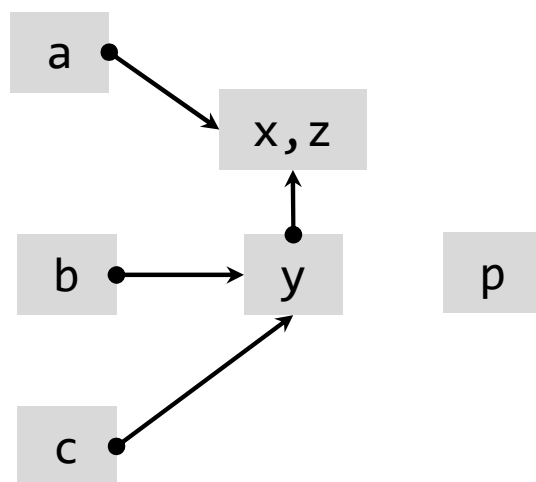
```
a = &x  
b = &y;  
if p  
    y = &z;  
else  
    y = &x;  
c = &y;
```



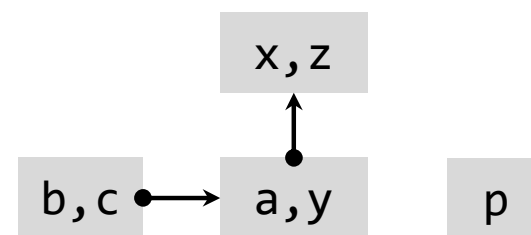
```
a, &x  
b, &y;  
p  
y, &z;  
y, &x;  
c, &y;
```



Andersen



Steensgaard



纯并查集（简化版）

Andersen vs Steensgaard

- 都是flow-insensitive、context-insensitive
- 不同点在于points-to集合的构造思路
 - Andersen-style:
 - 基于子集关系的
 - 每个节点对应一个变量
 - 每个节点有多条出边
 - 比较精准但效率不高
 - Steensgaard-style:
 - 基于等价关系的
 - 每个节点对应多个变量
 - 每个节点只有一条出边
 - 比较快但精准度有限

Flow-insensitive分析的缺陷

- 相对流敏感算法分析结果不够精准，可用性较差
 - 代码优化
 - 漏洞检测
- 如何进行流敏感、路径敏感的指针分析？
 - 维护每个program point的alias关系？

```
int a = 1;
int b = 2;
int c = 3;

int x = &a;
doSth();
x = &b;
if(x == &c) // 恒为假
    doSth();
if(x == &a) // 恒为假
    doSth(x)
```

```
int* a = malloc(10);
int* b = malloc(10);

int* x = a;
doSth();
x = b;
free(a) // a和x不是alias
free(x)
```

五、数据流分析进阶

回到常量分析问题

- 下列哪个程序会使基于lattice的常量检测算法失准？

```
x = 1;  
if (x==2)  
    x = 3;  
x是否为常量?
```

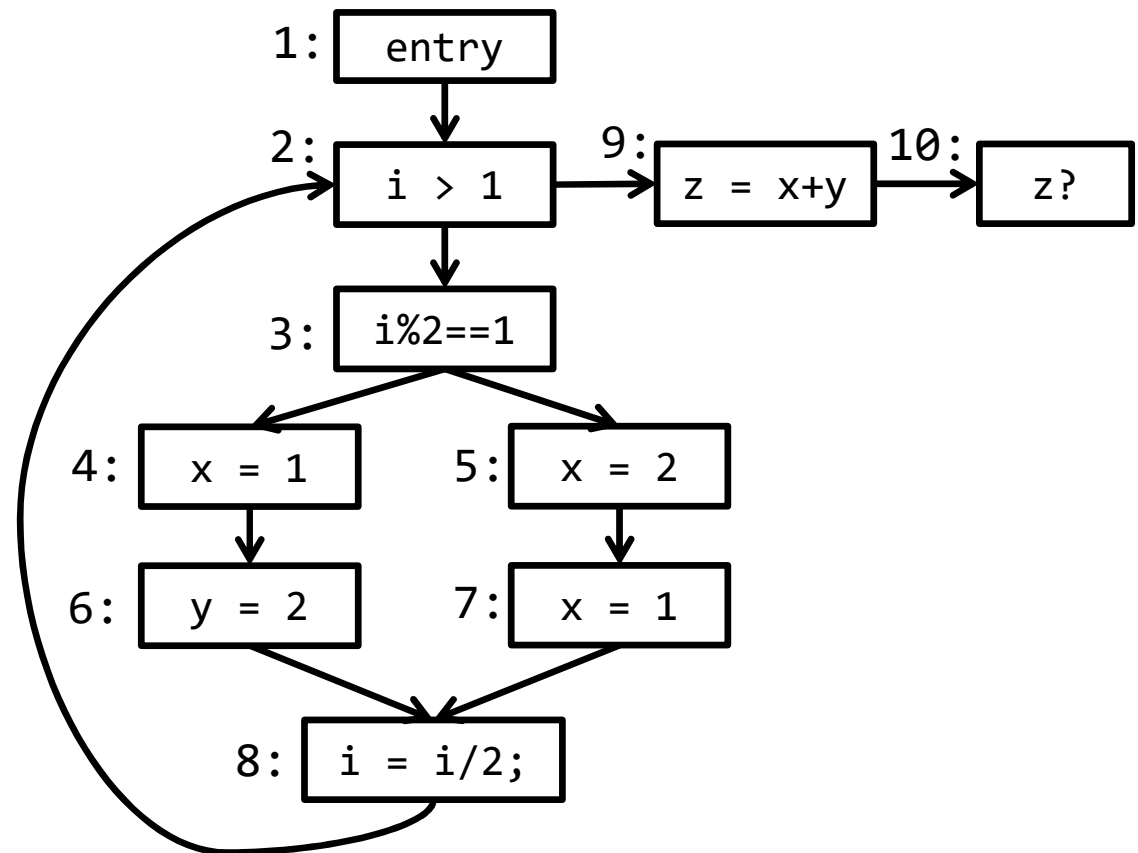
```
if (p)  
    x = 1;  
else  
    x = -1;  
y = x*x;  
y是否为常量?
```

```
if (p)  
    x = 1;  
else  
    x = 2;  
if (p)  
    y = x+2;  
else  
    y = x+1;  
y是否为常量?
```

Meet/Join Over All Passes

- 一个program point的属性信息是由每一条到达该点的路径分别计算得到的。
- 主要挑战：循环导致路径数是无限的

```
while (i>1) {  
  if (i%2==1){  
    x = 1;  
    y = 2;  
  } else {  
    x = 2;  
    y = 1;  
  }  
  i = i/2;  
}  
z = x+y;  
z是否为常量?
```



是否可以缩环？

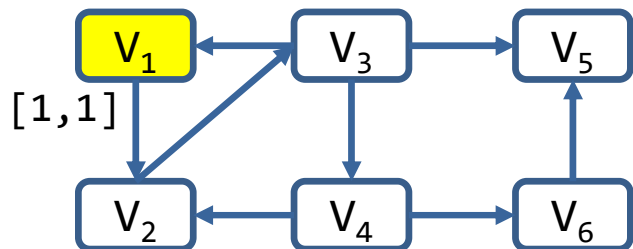
- 是否可以基于强联通分量缩环？
 - 检测强联通分量：Tarjan算法

```
DFSVisit(v)
{
    N[v] = c; //记录每个节点的到达时间
    L[v] = c; //记录下一跳的最早到达时间
    c++;
    push v onto the stack;
    for each w in OUT(v) {
        if N[w] == UNDEFINED {
            DFSVisit(w);
            L[v] = min(L[v], L[w]);
        } else if w is on the stack {
            L[v] = min(L[v], N[w]);
        }
    }
    if L[v] == N[v] { //找到强联通分量
        pop vertices off stack down to v;
    }
}
```

Tarjan算法找SCC: DFS

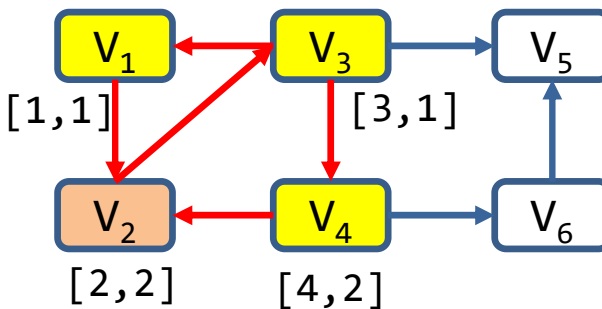
Stack:

V1[1,1]

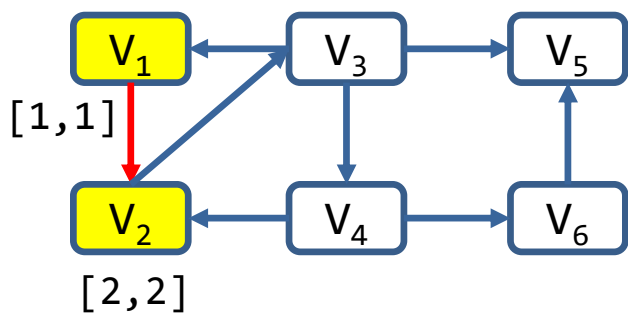


Stack:

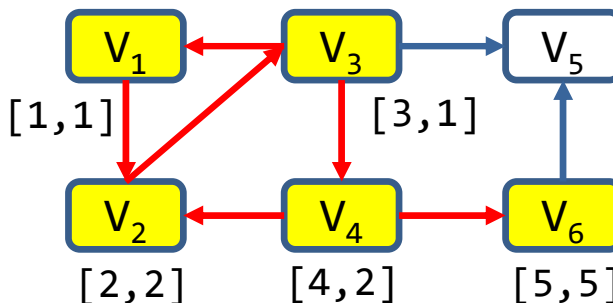
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



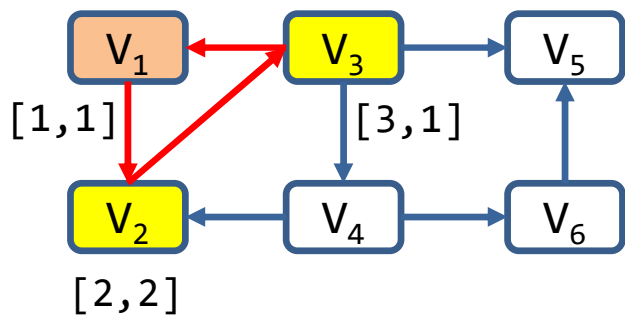
V2[2,2]
V1[1,1]



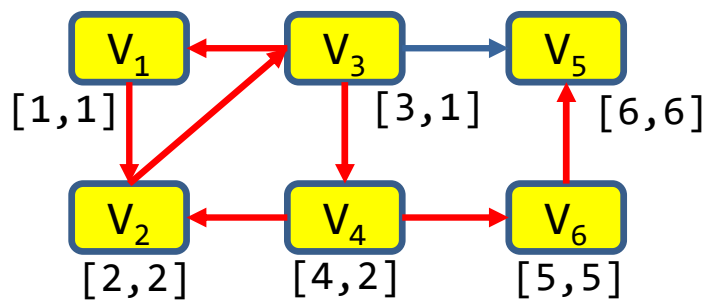
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



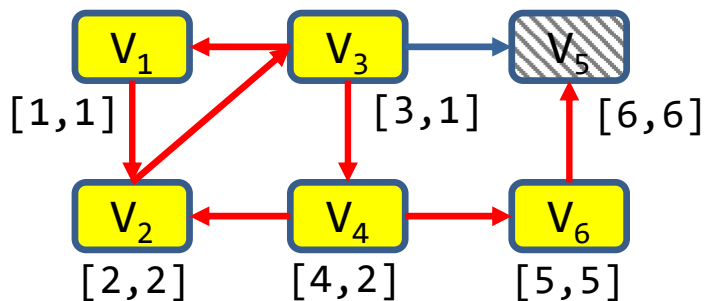
V3[3,1]
V2[2,2]
V1[1,1]



V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



通过Tarjan算法找SCC

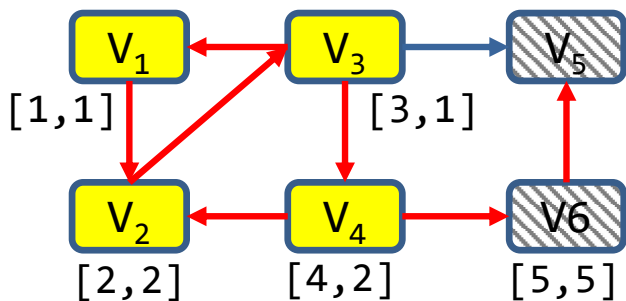


Stack:

V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

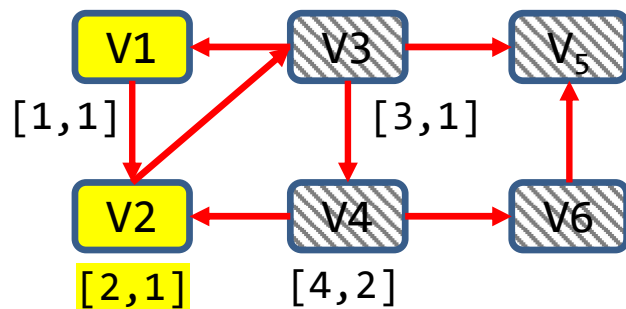
SCC:

{V5}



V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

{V5}
{V6}



$\min(L[v], L[w]);$

V2[2,1]
V1[1,1]

{V5}
{V6}
{4,3,2,1}

课后阅读

- 《编译原理（第2版）》
 - 第9章: Machine Independent Optimizations
- 《编译器设计（第2版）》
 - 第8章: 优化简介
 - 第9章: 数据流分析
 - 第10章: 标量优化

参考资料

- B.Kam and J.D.Ullman,, Monotone data flow analysis frameworks. Acta informatica, 1977.
- [Andersen'94] Andersen, Lars Ole. "Program analysis and specialization for the C programming language." PhD diss., University of Copenhagen, 1994.
- Steensgaard, Bjarne. "Points-to analysis in almost linear time." *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.
- [Hind'01] Michael Hind, Pointer analysis: Haven't we solved this problem yet?. In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering*, 2001.
- Whaley, John, and Monica S. Lam. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams." *ACM SIGPLAN Notices* 39.6 (2004): 131-144.
- Hardekopf, Ben, and Calvin Lin. "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code." *ACM SIGPLAN Notices*. Vol. 42. No. 6. ACM, 2007.
- Bravenboer, Martin, and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses." *ACM SIGPLAN Notices*. Vol. 44. No. 10. ACM, 2009.
- <http://web-static-aws.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf>.
- <https://www.cs.cmu.edu/afs/cs/academic/class/15745-s16/www/lectures/L6-Foundations-of-Dataflow.pdf>