《软件安全漏洞分析与发现》

程序切片

闫佳
中国科学院软件研究所
2024年3月20日

内容提纲

（ChatGPT Q&A): 程序切片对于发现漏洞有用吗？

Is the technique of program slicing beneficial in vulnerability discovery

Yes, program slicing can be beneficial in vulnerability discovery because it helps to identify the portions of code that are relevant to a particular behavior or property. Program slicing is a technique that extracts the code statements that may have an impact on a particular variable or condition. By identifying the relevant code statements, program slicing can help developers and security researchers to understand the flow of the code and the dependencies between different parts of the code.

In the context of vulnerability discovery, program slicing can be used to identify the code that is relevant to a particular security issue or vulnerability. For example, if a security researcher is trying to identify a buffer overflow vulnerability in a program, program slicing can be used to identify the code statements that manipulate the buffer and the variables that control the buffer size. This can help the researcher to understand how the vulnerability is triggered and how it can be exploited.

Overall, program slicing can be a useful tool for vulnerability discovery, as it can help to reduce the complexity of the code and focus on the parts that are most relevant to the security issue at hand.

内容提纲

- 概述
- 基本原理和组成
  - 数据流、控制流、程序依赖图
- 静态程序切片
  - 基于数据流方程的切片方法
  - 基于图可达性分析的切片方法
- 动态程序切片
  - 基于程序依赖图的切片方法
  - 基于动态依赖图的切片方法
- 切片工具介绍
- 小结

1、概述

- 程序切片
  - 旨在从程序中提取满足一定约束条件的代码片段，是一种用于分解程序的程序分析技术
- 应用
  - 软件理解和调试
    - 通过切片实现程序分解和快速错误定位
  - 软件维护和测试
    - 确定局部代码修改对软件其他部分的影响，从而降低回归测试中需要重新实施的测试用例数量
  - 软件逆向和安全性分析
    - 程序数据结构逆向
    - 漏洞分析

- 典型程序切片示例（后向切片）

```
1:    int main() {
2:        int x, y, z;
3:        int i=0;
4:        z = 0;
5:        y = getchar();
6:        for(; i<100;i++)
7:            if (i%2 == 1)
8:                x += y*i;
9:            else
10:               z += 1;
11:       printf("%d\n",x);
12:       printf("%d\n",z);
13:   }
```

源程序

(11 , x)
(语句编号，变量名)

切片后

```
1:    int main() {
2:        int x, y, z;
3:        int i=0;
4:        z = 0;
5:        y = getchar();
6:        for(; i<100;i++)
7:            if (i%2 == 1)
8:                x += y*i;
9:            else
10:               z += 1;
11:       printf("%d\n",x);
12:       printf("%d\n",z);
13:   }
```

切片

- 典型程序切片示例（后向切片）

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
4:          z = 0;
5:          y = getchar();
6:          for(; i<100;i++)
7:              if (i%2 == 1)
8:                  x += y*i;
9:              else
10:                 z += 1;
11:         printf("%d\n",x);
12:         printf("%d\n",z);
13:     }
```

源程序

(11 , x)

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
5:          y = getchar();
6:          for(; i<100;i++)
7:              if (i%2 == 1)
8:                  x += y*i;
11:         printf("%d\n",x);
13:     }
```

(12 , z)

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
6:          for(; i<100;i++)
7:              if (i%2 == 1)
9:              else:
10:                 z += 1;
11:         printf("%d\n",z);
13:     }
```

1、概述

- 切片准则
  - < S , V >
    - S: 语句（或指令）标识
    - V: 变量
  - 程序P的关于切片准则<S,V>的切片 Slice<S,V> :

由P中所有影响变量V的语句或者受变量V影响的语句组成

- 切片性质：
  - 切片Slice<S,V> 通过删除P中语句而得到
  - 切片中语句的语义行为与源程序保持一致

1、概述

- 获取最小切片的算法
  - 针对切片准则<6,a>的切片是否包含第5行代码
    - 最小切片算法必须判断每一条语句是否包含在切片中
  - 上述问题等价于图灵停机问题
    - 反证法
  - 最小切片算法不存在

```
1:  int a,b;
2:  a = getchar();
3:  b = getchar();
4:  if (a > b)
    {
            $%%%&*
5:          a = a*b
     }
6:   printf(a);
```

1、概述

- 从程序分析方法视角看“程序切片”

程序分析的理想目标：

证明程序的运行满足性质 ϕ

性质ϕ：各类程序分析问题

- 基础问题：切片、到达定义、变量活性
- 应用问题：软件漏洞（缓冲区溢出、内存泄漏

Rice定理：递归可枚举语言的任何非平凡的性质，都是不可判定的

- 递归可枚举语言：现有的程序设计语言C、C++、Java
- 平凡性质：任何程序都满足的性质
- 非平凡性质：有些程序满足，有些程序不满足

1、概述

- 从程序分析方法视角看“程序切片”

程序分析的现实目标：

① 对程序进行抽象，形成可判定的模型M

② 判定模型M是否满足性质 φ

Truth：不可判定的

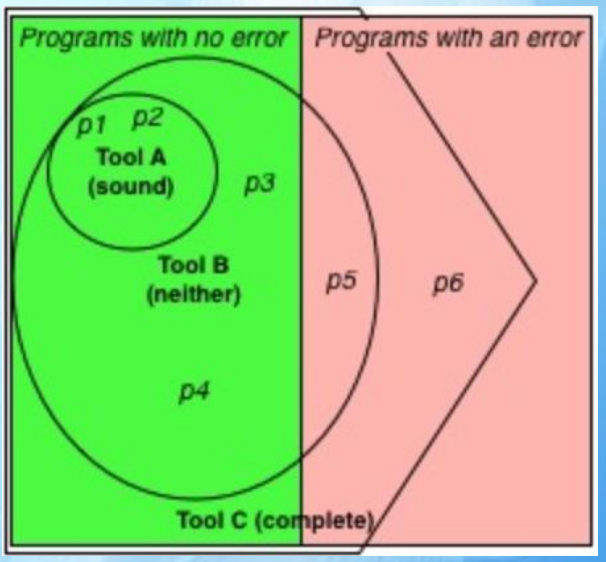
Sound：

如果程序分析算法声称某程序满足性质φ，则该程序确实满足性质φ

如有图，性质φ为程序不存在bug

Tool A 判定程序p1,p2满足性质φ，p1,p2确定满足该性质，不存在bug

Tool A 认为 p3, p4, p5, p6 存在bug，其中p3,p4 为误报



http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/

1、概述

- 从程序分析方法视角看“程序切片”

程序分析的现实目标：

① 对程序进行抽象，形成可判定的模型M

② 判定模型M是否满足性质 φ

Truth：不可判定的

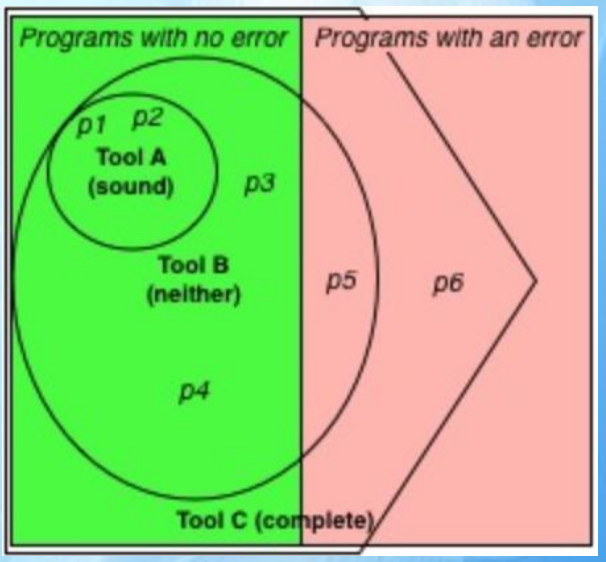
Unsound

如果程序分析算法声称某程序满足性质φ，但存在反例能证明该程序不满足性质φ

如有图，性质φ为程序不存在bug

Tool B 判定程序p1,p2,p3,p4满足性质φ，p1,p2,p3,p4确定满足该性质，不存在bug

Tool B 判定程序p5满足性质φ，p5实际上不满足该性质，存在bug



http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/

- 从程序分析方法视角看“程序切片”

程序分析的现实目标：

① 对程序进行抽象，形成可判定的模型M

② 判定模型M是否满足性质 φ

Truth：不可判定的

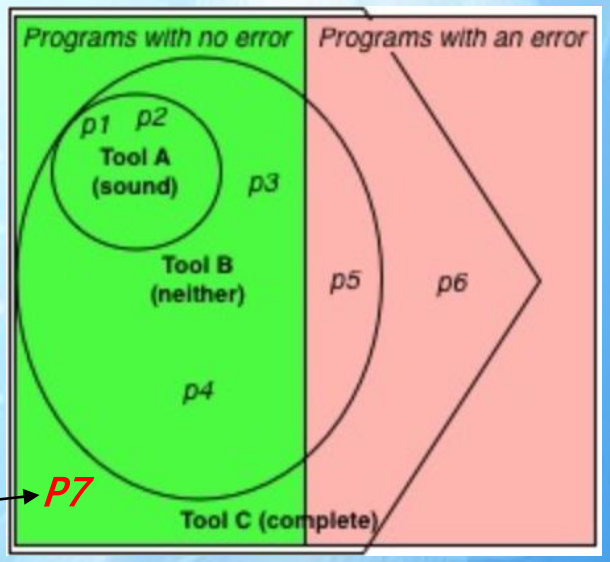
Complete：

找到的不满足性质φ的所有情况都是正确的，不保证找到所有不满足φ的情况

如有图，性质φ为程序不存在bug

Tool C 判定程序p1,p2,p3,p4满足性质φ，p1,p2,p3,p4确定满足该性质，不存在bug

Tool C 判定程序p5,p6满足性质φ，p5,p6实际不满足该性质，存在bug



http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/

1、概述

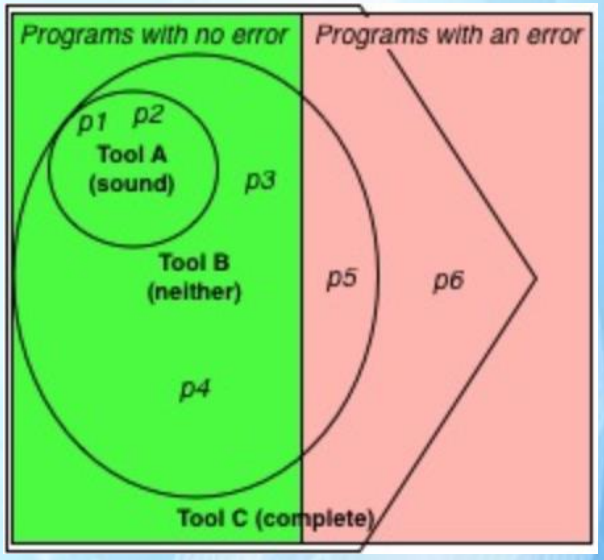
- 从程序分析方法视角看“程序切片”

程序切片是 Sound 优先

性质 φ：哪些语句不在切片里
切片中可能包含并不相关的代码/语句

大多数程序分析方法是 Sound 优先

很强的保证：当模型定义良好时，可找到程序中所有不满足性质 φ 的情况



Complete 优先的情况

只要判断某个程序不满足性质 φ（缓冲区溢出），则该程序一定不满足
举例：基于符号执行挖掘漏洞，如KLEE

http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/

1、概述

TCA

- 分类
  - 程序输入对切片的影响
    - 静态切片 → 有条件切片 → 动态切片
  - 影响指定变量还是被指定变量影响
    - 前向切片和后向切片
  - 切片对象
    - 软件规约切片、源程序代码切片、二进制代码切片
  - 是否可执行
    - 可执行程序切片和不可执行程序切片

1、概述

the categories of program slicing algorithm

Program slicing is a technique used in software engineering to extract relevant portions of code related to specific functionality or behavior. There are different types of program slicing algorithms, and they can be categorized as follows:

1. Static program slicing: This type of program slicing is performed at the source code level and does not involve the execution of the program. It analyzes the control and data flow of the program to identify the statements that contribute to a particular slice. Static slicing algorithms include data flow analysis, control flow analysis, and dependence-based slicing.
2. Dynamic program slicing: This type of program slicing is performed at runtime and involves the execution of the program. It identifies the statements that contribute to a slice based on the values of program variables at runtime. Dynamic slicing algorithms include execution-based slicing, trace-based slicing, and backward slicing.
3. Hybrid program slicing: This type of program slicing combines static and dynamic slicing techniques to achieve better results. It uses static analysis to identify potential program paths and then performs dynamic slicing to refine the slice. Hybrid slicing algorithms include forward and backward hybrid slicing.
4. Incremental program slicing: This type of program slicing updates the slice incrementally as the program changes. It is useful in large software systems where changes are frequent, and it can save time and effort by avoiding complete slicing after every change.
5. Context-sensitive program slicing: This type of program slicing takes into account the context of the program, such as the calling context, to identify relevant statements for a slice. It is particularly useful in object-oriented programming languages.

Overall, program slicing algorithms can help developers understand the behavior of their code, debug it more efficiently, and maintain it more effectively.

- 分类
  - 静态切片
  - 动态切片
  - 混合切片
  - 增量切片
  - 上下文相关切片

1、概述

- 研究发展历史
  - 基于数据流方程的方法
    - 静态后向切片、可执行切片、过程内切片
  - 基于程序依赖图的方法
    - 静态、动态切片
  - 面向对象程序设计
    - C++， Java
  - 面向不同应用的各类切片计算
    - 削片、砍片、波动分析
    - 软件逆向工程、软件维护、程序验证
    - 软件安全（软件漏洞挖掘、错误定位和修复）

2、基本原理和组成

TCA

- 控制流分析
  - 基本块
    - 程序语句的基本模块划分
  - 控制流图
    - 程序基本模块之间的控制跳转
- 数据流分析
  - 可到达定义：变量定义
  - 活性：变量引用
- 程序依赖图
  - 程序数据流和控制流的有向图表示

2、基本原理和组成 – 控制流分析

TCA

- 控制流分析
  - –分析程序执行流程：通常为静态分析

程序语言	汇编语言	C语言	C++语言
顺序结构			
条件结构	jz jnz je jne …	if else	if else
循环结构	loop, rep	while, for	while,for
函数调用结构	call	void func(void)	void A::func(void)
其他	jmp	goto	虚函数

2、基本原理和组成 – 控制流分析

TCA

- 基本块
  - 连贯语句构成，程序从第一条语句进入，从最后一条语句离开，拥有唯一入口和出口
- 基本块特点
  - 1. 程序执行时只能从该基本块的第一条指令进入该基本块
  - 2. 程序执行时离开该基本块前的最后一条指令必须是该基本块的最后一条指令
- 基本块有什么用
  - 最基础的模块划分：控制流分析的最基础代码单元

算法：基本块计算方法

输入： I = { ins_i,i∈{1,2,3,…,n}}指令序列

```
LeaderSet // 基本块首条指令集合
BlockSet -> {  }// 基本块集合
Begin
    For ins in I:
        If ins 是分支开头指令
            LeaderSet = LeaderSet ∪{ ins }
    For x in LeaderSet:
        BlockSet[x] = {x}
        i = x + 1
        while i<=n and (not i∈LeaderSet)
            BlockSet[x] = BlockSet[x] ∪ {i}
            i = i + 1
        End
```

LeaderSet
- 程序入口
- 跳转目标语句（指令）
- 跳转的后一条语句（指令）

2、基本原理和组成 – 控制流分析

TCA

■ 基本块计算示例

```
1:  int a,b;
2:  a = getchar();
3:  b = getchar();
4:  if (a > b)
    {
5:          a = a*b
6:          printf(a);
    }
7:  else
    {
8         a = a/b
9:        printf(a);
    }
```

LeaderSet = { 1, 5 , 8 }

BlockSet[1] = { 1 }

BlockSet[1] = { 1 , 2 }

BlockSet[1] = { 1 , 2 , 3 }

BlockSet[1] = { 1 , 2 , 3 , 4 }

BlockSet[2] = { 5 }

BlockSet[2] = { 5 , 6 }

BlockSet[3] = { 8 }

BlockSet[3] = { 9 }

2、基本原理和组成 – 控制流分析

■ 基本块计算示例

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a,b;
    a = getchar();
    b = getchar();
    if (a > b)
    {
        a = a*b;
        printf("%u\n",a);
    }
    else
    {
        a = a/b;
        printf("%u\n",a);
    }
}
```

Code Comparison

```
int main(void)
{
    int a,b;
    a = getchar();
    b = getchar();
    if (a > b)
    {
        a = a*b;
        printf("%u\n",a);
    }
    else
    {
        a = a/b;
        printf("%u\n",a);
    }
}
```

```
0000 55                 push    rbp
                        .cfi_def_cfa_offset 16
                        .cfi_offset 6, -16
0001 4889E5             mov     rbp, rsp
                        .cfi_def_cfa_register 6
0004 4883EC10           sub     rsp, 16
0008 B8000000           mov     eax, 0
     00
000d E8000000           call    getchar
     00
0012 8945F8             mov     DWORD PTR [rbp-8], eax
0015 B8000000           mov     eax, 0
     00
001a E8000000           call    getchar
     00
001f 8945FC             mov     DWORD PTR [rbp-4], eax
0022 8B45F8             mov     eax, DWORD PTR [rbp-8]
0025 3B45FC             cmp     eax, DWORD PTR [rbp-4]
0028 7E20               jle     .L2
                .LBB2:
002a 8B45F8             mov     eax, DWORD PTR [rbp-8]
002d 0FAF45FC           imul    eax, DWORD PTR [rbp-4]
0031 8945F8             mov     DWORD PTR [rbp-8], eax
0034 8B45F8             mov     eax, DWORD PTR [rbp-8]
0037 89C6               mov     esi, eax
0039 BF000000           mov     edi, OFFSET FLAT:.LC0
     00
003e B8000000           mov     eax, 0
     00
0043 E8000000           call    printf
     00
                .LBE2:
0048 EB1E               jmp     .L3
                .L2:
                .LBB3:
004a 8B45F8             mov     eax, DWORD PTR [rbp-8]
004d 99                 cdq
004e F77DFC             idiv    DWORD PTR [rbp-4]
0051 8945F8             mov     DWORD PTR [rbp-8], eax
0054 8B45F8             mov     eax, DWORD PTR [rbp-8]
0057 89C6               mov     esi, eax
0059 BF000000           mov     edi, OFFSET FLAT:.LC0
     00
005e B8000000           mov     eax, 0
     00
0063 E8000000           call    printf
     00
```

现实情况下二进制程序的基本块识别

- 静态分析无法识别由于间接跳转形成的基本块
  - 函数指针： void (*func_A)(char *)
  - 间接跳转： jmp [eax]
  - 返回地址修改
- 异常处理
  - 可预期异常 try catch
  - 非预期异常
    - 内核、IO等

```
call    sub_30CA6D74
sub     esp, 408h
add     esp, 8
popa
popf
pop     ebp
add     esp, 28h
lea     ebx, loc_30CC88FB
mov     off_31600CC8, ebx
popa
popf
```

```
...
; esi holds the ret location
mov     eax, [esi]
add     eax, 0Ch
mov     [esi], eax
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn
```

Office 2003运行时修改栈上的返回地址

```
; __int64 __usercall sockwrite@<rax>(int fd@<edi>, char
                public sockwrite
sockwrite       proc near               ; CODE XREF: pa
                                        ; login+4D↓p ..

arg             = __va_list_tag ptr  800h
var_s820        = byte ptr  820h
var_s830        = qword ptr  830h
var_s838        = qword ptr  838h
var_s840        = qword ptr  840h
var_s848        = qword ptr  848h
var_s8CF        = byte ptr  8CFh
arg_0           = byte ptr  8F0h

; __unwind {
                push    rbp
                movzx   eax, al
                push    rbx
                mov     ebx, edi
                sub     rsp, 8D8h
                mov     [rsp+var_s830], rdx
                lea     rdx, ds:0[rax*4]
                mov     eax, offset loc_402256
                mov     [rsp+var_s838], rcx
                mov     [rsp+var_s840], r8
                lea     rcx, [rsp+arg]  ; arg
                sub     rax, rdx
                lea     rdx, [rsp+var_s8CF]
                mov     [rsp+var_s848], r9
                mov     rdi, rsp        ; s
                mov     rbp, rsp
                jmp     rax
                movaps  xmmword ptr [rdx-0Fh], xmm7
```

异常处理

TCA

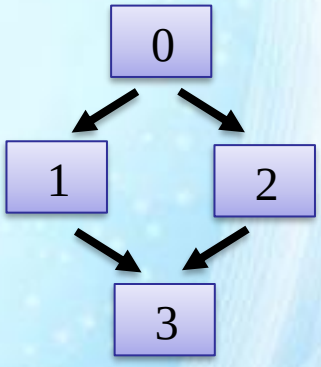
■控制流图（CFG）

– 以基本块为节点，依据程序控制流连接各个节点构成的有向图

```
1:    int max(int x, int y) {
2:      int t = 0;
3:      if (x > y)
4:            tmp = x;
5:      else
6:            tmp = y;
7:      return tmp;
8:    }
```

基本块划分

```
···· 0
···· 0
···· 1

···· 2
···· 3
```



构建CFG

2、基本原理和组成 – 控制流分析

TCA

算法：控制流图构造算法

输入：

BlockList // 基本模块列表

BranchMap branchInst -> {Target(branchInst)} // 跳转指令到跳转目标集合的映射

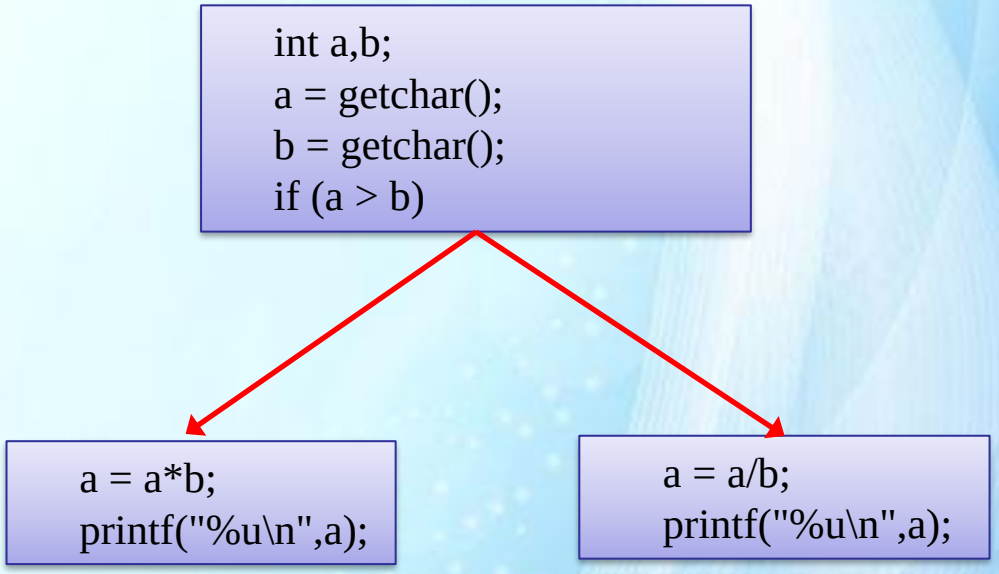
输出：

CFG // 控制流图

```
Begin
    For B in BlockList:
        x = B[len{B}-1]
        if x ∈ BranchMap.keySet
            for B_Target in BranchMap[x]:
                create_edge(B,B_Target);
        if not x ∈ BranchMap.keySet
            create_edge(B, Next(B));
End
```

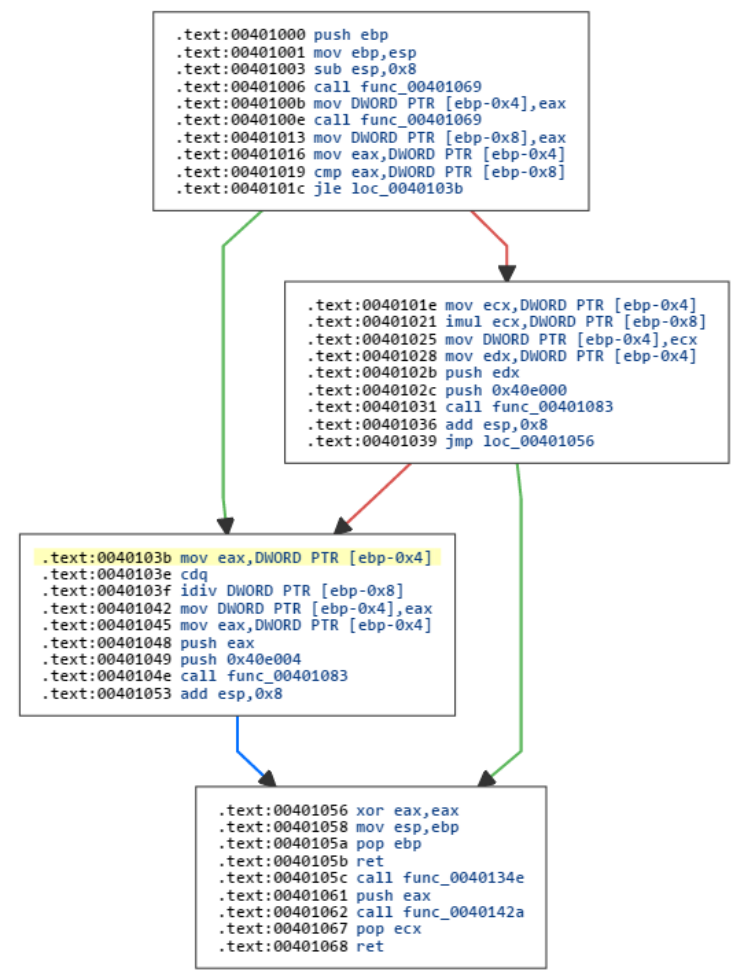
控制流图示例

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a,b;
    a = getchar();
    b = getchar();
    if (a > b)
    {
        a = a*b;
        printf("%u\n",a);
    }
    else
    {
        a = a/b;
        printf("%u\n",a);
    }
}
```



控制流图示例 – 汇编代码

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a,b;
    a = getchar();
    b = getchar();
    if (a > b)
    {
        a = a*b;
        printf("%u\n",a);
    }
    else
    {
        a = a/b;
        printf("%u\n",a);
    }
}
```



2、基本原理和组成 – 控制流分析

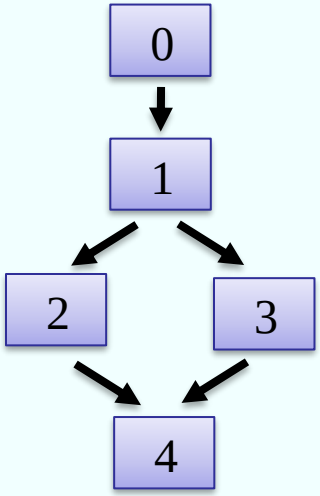
TCA

- 控制流图（简称“CFG”）
  - 四元组，G={ V,E, Entry,Exit }
    - V是基本块节点的集合
    - E是基本块之间边的集合
    - Entry表示入口节点，Exit表示结束节点
  - 前驱和后继节点
    - 存在有序对<a,b>∈E，其中a,b∈V，则称a是b的直接前驱，b是a的直接后继
  - 可执行路径
    - 从CFG中某个节点开始进行图遍历所形成的路径

2、基本原理和组成 – 控制流分析

- 控制流图（简称“CFG”）
  - 前必经节点
    - 如果节点a,b满足下列条件，则称节点a是节点b的前必经节点，记为a->b
      - 从节点Entry到节点b的所有路径都经过节点a
    - 若a->b，也称a支配节点b
  - 严格前必经节点
    - 节点a，b，有a->b，且a!=b，则称a是b的严格前必经节点，记为$a\text{->}_p b$
  - 直接前必经节点
    - 节点a，b，有a->b，满足下列条件，则a是b的直接前必经节点
      - 不存在节点c，满足$a\text{->}_p c$，同时$c\text{->}_p b$
  - 后必经节点，严格后必经节点，直接后必经节点

- 控制流图示例



节点 0 是 节点 1 的严格直接前必经节点

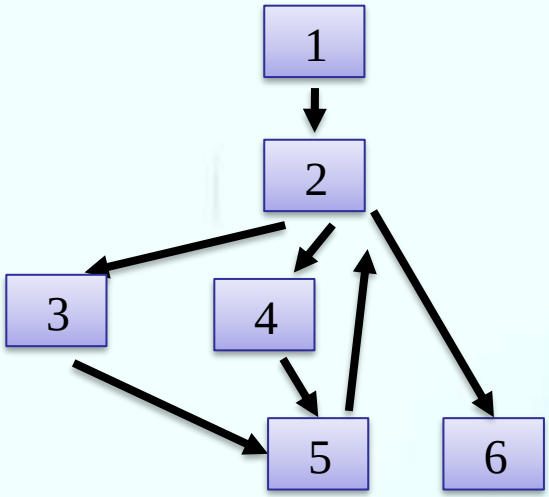
节点 0 是 节点 2 的严格前必经节点

节点 4 是 节点 2 的严格直接后必经节点

节点 4 是 节点 1的严格后必经节点

- 控制流图示例

节点？ 是 节点 5 的严格直接前必经节点



????

2、基本原理和组成 – 数据流分析

TCA

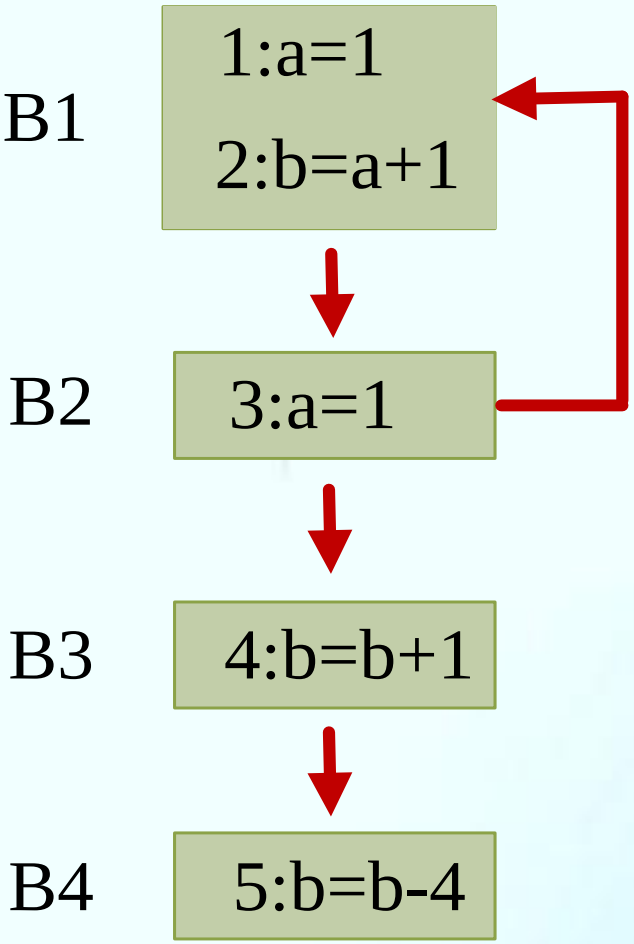
- 数据流分析
  - 定义一：
    - 收集计算机程序在不同点计算的值的信息的技术
  - 定义二：
    - 在程序一定范围内，确定变量定值和引用间关系的工作
  - 定义三：
    - 研究变量如何从定义赋值点流动到它的使用点，继而又被重新赋值定义的过程
- 目标
  - 针对语句 n中 的变量 v：
    - **可到达定义分析**：变量v的值是哪里定义的
    - **活性分析**：变量v将会被哪条语句所使用

2、基本原理和组成 – 数据流分析

TCA

- 可到达定义
  - “定义”：
    - 变量的“**定义**”：程序中对该变量进行赋值/改写的语句
  - 可到达定义：
    - 变量x的一个“定义”达到 p，当前仅当在程序控制流图（CFG）中存在从该“定义”对应的节点到 p 节点的一条路径，并且该路径上没有变量x的其他“定义”

- 可到达定义示例



a 的定义是 语句 1，3

b 的定义是 语句 2，4，5

语句1的定义可到达语句2

语句2的定义可达到语句3、4

语句3的定义可到达语句4、5

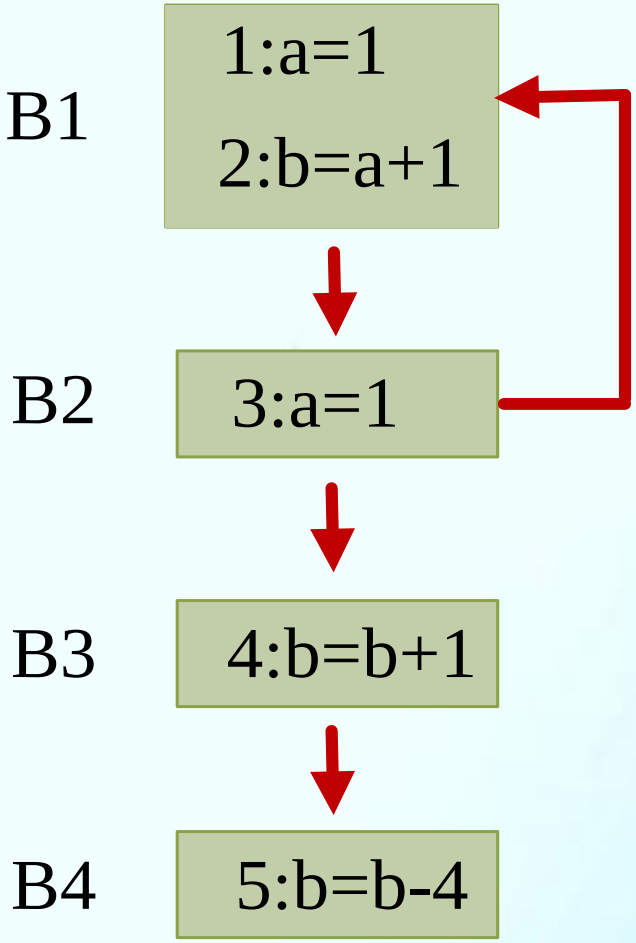
2、基本原理和组成 – 数据流分析

TCA

- 可到达定义分析 – 变量定义和引用相关的集合定义
  - 前提：针对某条特定语句S和变量x
  - 针对变量x有
    - 定义集 Def(x)：定义x的所有语句的集合
    - 引用集Ref(x)：引用x的所有语句的集合
  - 针对语句S有：
    - 产生集Gen(S)：语句S给出的变量定义所在语句的集合
    - 消灭集Kill(S)：语句若S重新定义变量x，而x此前由语句S’定义，则称S消灭定义S’。所有由S消灭的定义的集合称为S的消灭集
    - 入集 In(S)：所有在语句S之前仍然有效（没有被消灭）的定义语句的集合
    - 出集 Out(S)：所有离开语句S的定义语句的集合：包括当前语句的入集的节点，以及产生集的节点，同时去掉语句S所消灭的定义语句

2、基本原理和组成 – 数据流分析

- 可到达定义分析 – 变量定义和引用相关的集合定义
  - 示例



Def(a) = { 1,3 }

Use(a) = { 2 }

Gen(2) = { 2 }

Kill(2) = {4，5 }

In(2) = { 1 }

Out(2) = { 1，2 }

Out(4) = { ? }

注意：Kill集是程序整体范围内当前语句的定义所杀死的定义的集合

算法：可到达定义计算

- **输入：** CFG G = { V,E, Entry,Exit }
- **输出：** Out

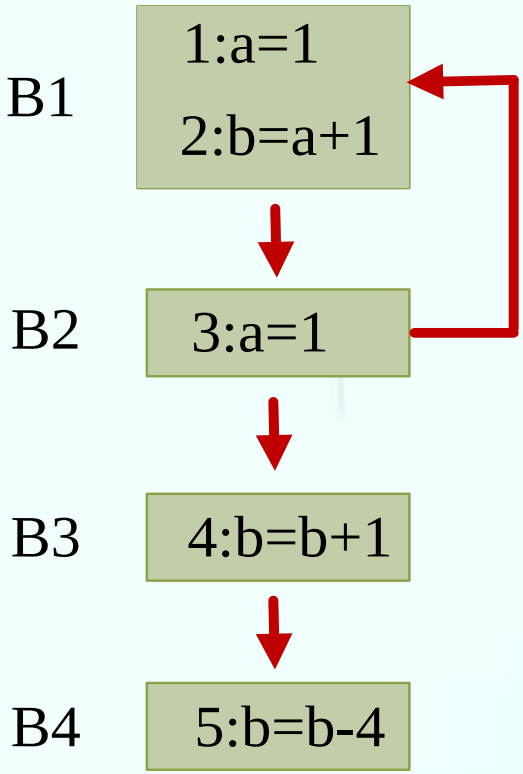
```
Begin
    For b in V:
        In(b) = φ
        Out(b) = φ
    Change = true
    while Change:
        Change = false
        for b in V:
            In (b) = ∪(out(p)) | for all p in Pred(b))
            OldOut = Out(b)
            Out(b) = Gen(b) ∪( In(b) – Kill(b) )
            If Out(b) != OldOut:
                Change = true
        End
```

一个节点的出集影响其后继节点的入集

Pred(b) 表示 b的前驱节点

- 可到达定义计算示例

$$Out(b) = Gen(b) \cup (In(b) - Kill(b))$$



G=Gen K=Kill I=In O=Out

	G_0	K_0	I_0	O_0	I_1	O_1	I_2	O_2	I_3	O_3	I_4	O_4
B1	1,2	3,4,5				1,2	3	1,2	2,3	1,2	2,3	1,2
B2	3	1				3	1,2	2,3	1,2	2,3	1,2	2,3
B3	4	2,5				4	3	3,4	2,3	3,4	2,3	3,4
B4	5	2,4				5	4	5	3,4	3,5	3,4	3,5

到第3轮迭代计算时，入集和出集收敛

注意：Kill集是程序整体范围内当前语句的定义所杀死的定义的集合

- 可到达定义计算示例

$$Out(b) = Gen(b) \cup (In(b) - Kill(b))$$

```
[0] c = input()
[1] var a = 6
[2] var b = a + 6
[3] if c == 3:
[4]     a = 4
[5]     c = a + 1
[6] else
[7]     a = 2
[8]     c = a + 2
[9] b = a + 2
```

[9]的可到达定义？

2、基本原理和组成 – 数据流分析

- 算法是否会终止：

  $Out(b) = Gen(b) \cup (In(b) - Kill(b))$

  - Gen 和 Kill 是固定的，In相同，则Out相同
  - 对于所有基本块来说，Out 是单调递增的
  - Out集的扩大是有上限的

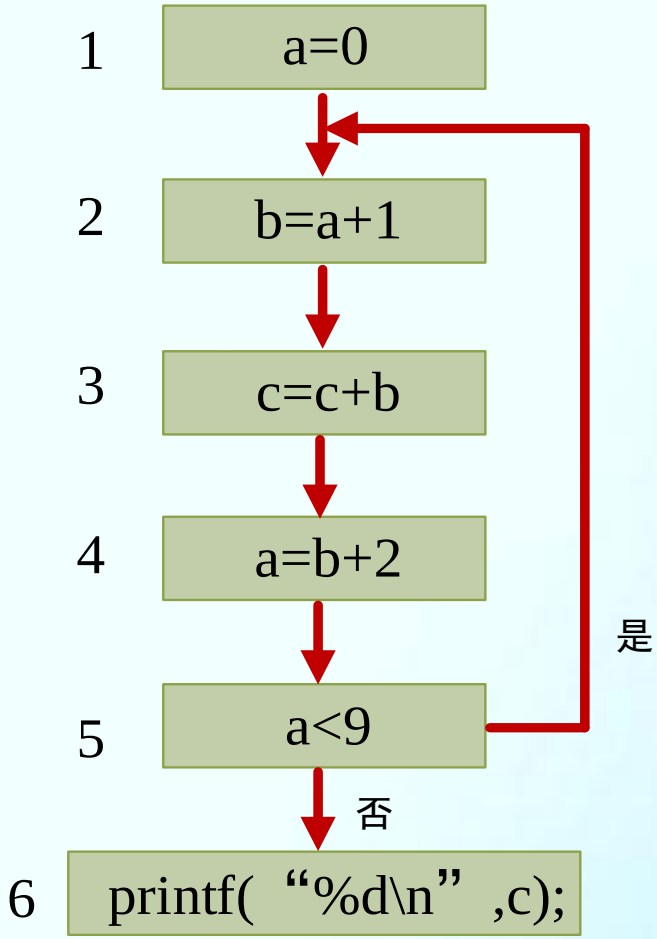
- 是否还有其他可到达定义计算方法
  - 路径遍历？
- 这个方法是否Sound，是否有误报？
  - 类似“最小切片”

2、基本原理和组成 – 数据流分析

TCA

- 活性分析
  - 定义：
    - 对某个语句定义的变量是否在后续语句中被引用，以及被那些语句引用的情况的分析
  - “活性”定义
    - 程序的某个点p和一个变量x，如果x在点p上的值会在控制流图中从p出发的某条路径上使用，就称x在p上活跃，否则称x在p上是死的
  - 实际应用：
    - 程序优化：寄存器分配
    - 漏洞利用？

- 活性分析示例



变量 a 在语句1是活的

变量 a 在语句3是死的

变量 b 在语句2是活的

算法：变量活性计算

- **输入：** CFG G = { V,E, Entry,Exit }
- **输出：** Out

```
Begin
    For b in V:
        In(b) = φ
        Out(b) = φ
    Change = true
    while Change:
        Change = false
        for b in V:
            Out (b) = ∪(In(p)) | for all p in Succ(b))
            OldIn = In(b)
            In(b) = Gen(b) ∪( Out(b) – Kill(b) )
            If In(b) != OldIn:
                Change = true
        End
```

一个节点的入集受其后继节点的出集的影响

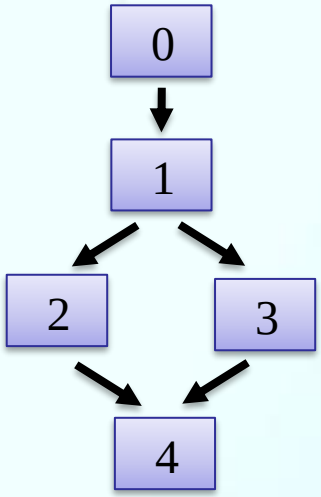
Succ(b) 表示 b的前驱节点

2、基本原理和组成 – 程序依赖图

- 程序依赖图
  - 控制依赖
    - 令G为某程序P的控制流图，其中a和b是G中的两个节点，当a和b满足下列两个条件，则称b->$_{cd}$a

1)从a到b有一条可执行路径，即CFG中节点a与节点b存在一条路径。同时对于该路径上除a，b外的每个节点n，节点b都是其后必经节点。

2)节点b不是a的后必经节点



节点 3 控制依赖于节点1

节点 1 控制依赖于节点0？ No

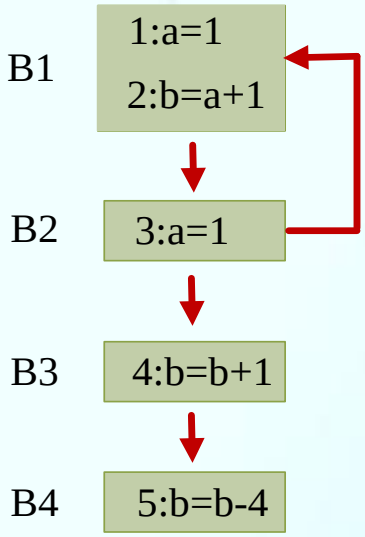
2、基本原理和组成 – 程序依赖图

- 程序依赖图
  - 数据依赖
    - 若a和b分别为程序P控制流图G中的两个节点，v为P中的一个变量，若a和b满足下列条件，则称b关于变量v直接数据依赖于a，记为b->$_{dp}$a

1）a对变量v进行定义，即v∈Def(a)

2）b中引用了变量v，即有v∈Use(b)

3）a到b有一条可执行路径，且在此路径上不存在语句对v进行定义



B3 数据依赖于 B1

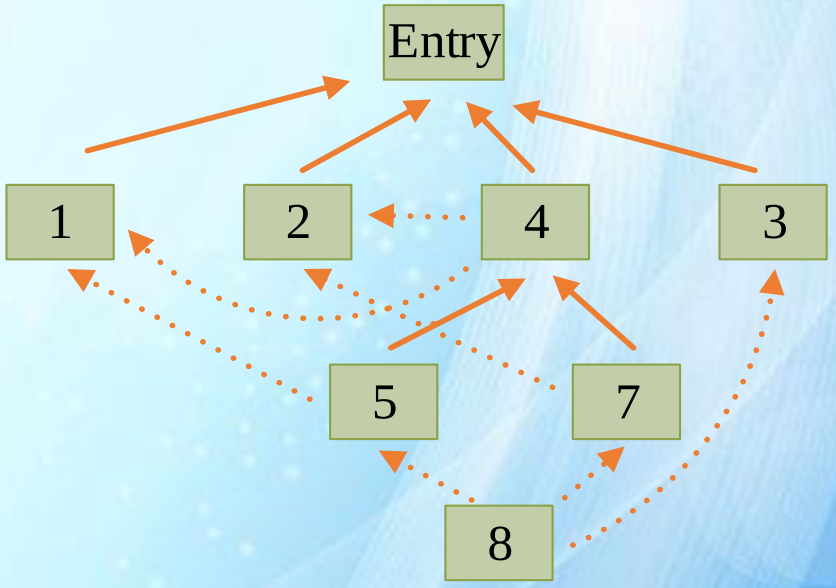
2、基本原理和组成

TCA

■ 程序依赖图

– PDG由程序的控制依赖图和数据依赖图组成。若$G_c = (V, C)$，$G_d = (V, D)$分别为程序P的控制依赖图和数据依赖图，则程序依赖图是$G_p = (V, E)$，其中$E = C \cup D \cup X$，其中X表示程序中的其他依赖关系

```
1:   int x = getch();
2:   int y = getch();
3:   int tmp = 0;
4:   if ( x > y)
5:       tmp = x;
6:   else
7:       tmp = y;
8:   printf( “%u\n” ,tmp);
```



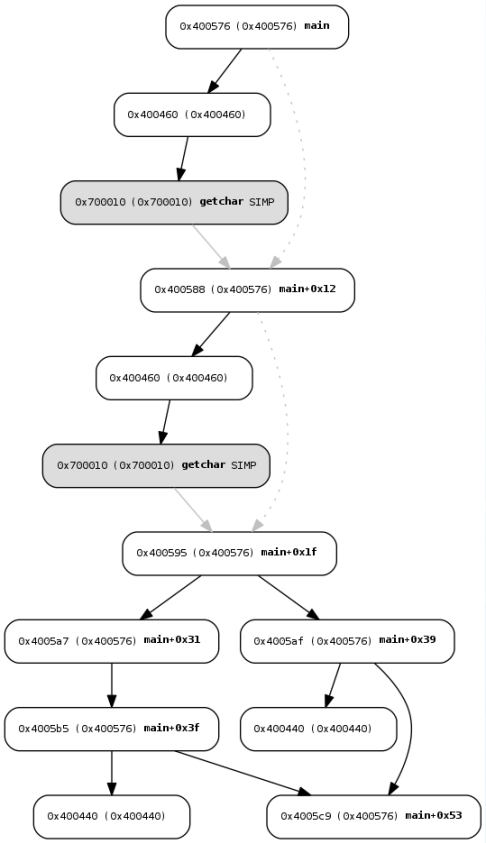
2、基本原理和组成

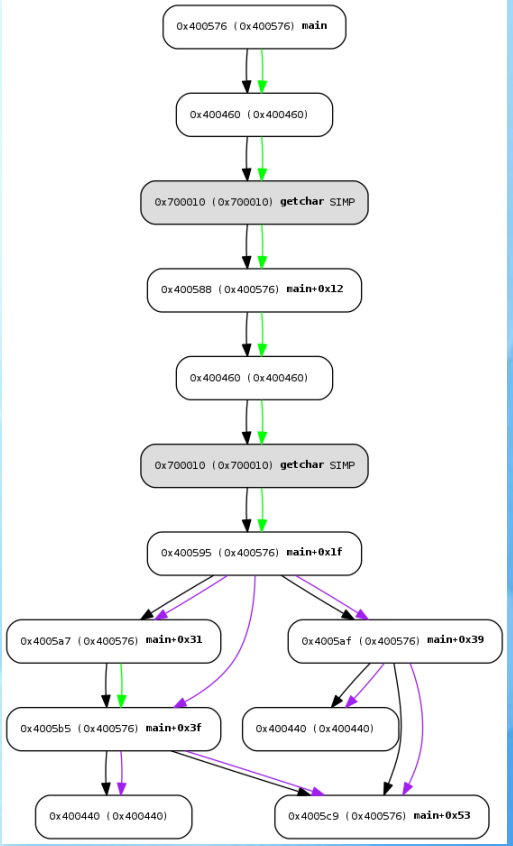
TCA

■ 程序依赖图

– PDG由程序的控制依赖图和数据依赖图组成。若$G_c = (V, C)$，$G_d = (V, D)$分别为程序P的控制依赖图和数据依赖图，则程序依赖图是$G_p = (V, E)$，其中$E = C \cup D \cup X$，其中X表示程序中的其他依赖关系

```
1:   int x = getch();
2:   int y = getch();
3:   int tmp = 0;
4:   if ( x > y)
5:       tmp = x;
6:   else
7:       tmp = y;
8:   printf( “%u\n” ,tmp);
```





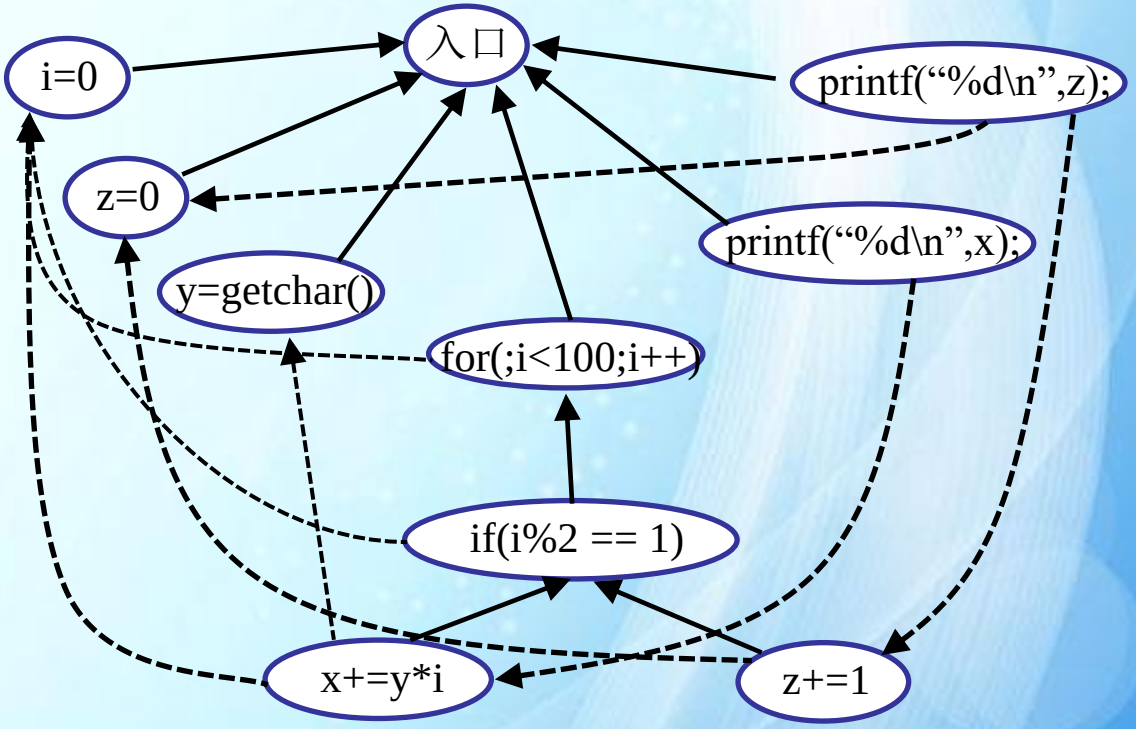
2、基本原理和组成

TCA

■程序依赖图

– 基于CFG构造控制依赖图和数据依赖图，从而形成程序依赖图

```
int main() {
    int x, y, z;
    int i=0;
    z = 0;
    y = getchar();
    for(; i<100;i++)
      if (i%2 == 1)
          x += y*i;
      else
          z += 1;
    printf("%d\n",x);
    printf("%d\n",z);
}
```



2、基本原理和组成

TCA

- 小结
  - 依赖关系
    - 控制依赖

      基本块（语句）之间的执行顺序关系

    - 数据依赖
      - 变量之间的定义和引用关系
      - 可到达定义
      - 活性分析
  - 程序依赖图
    - 囊括控制依赖和数据依赖的程序抽象表示

2、基本原理和组成

TCA

- 程序切片一般过程

程序依赖关系提取

- 基本块>>控制流图>>数据流方程
- 数据依赖，控制依赖
- 过程内依赖，过程间依赖，程序依赖图，系统依赖图

切片规则制定

- 静态切片、动态切片、有条件切片
- 前向切片、后向切片

切片生成

- 数据流方程
- 基于图可达性方法

- 典型程序切片示例（后向切片）

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
4:          z = 0;
5:          y = getchar();
6:          for(; i<100;i++)
7:              if (i%2 == 1)
8:                  x += y*i;
9:              else
10:                 z += 1;
11:         printf("%d\n",x);
12:         printf("%d\n",z);
13:     }
```

源程序

(11 , x)

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
5:          y = getchar();
6:          for(; i<100;i++)
7:              if (i%2 == 1)
8:                  x += y*i;
11:         printf("%d\n",x);
13:     }
```

(12 , z)

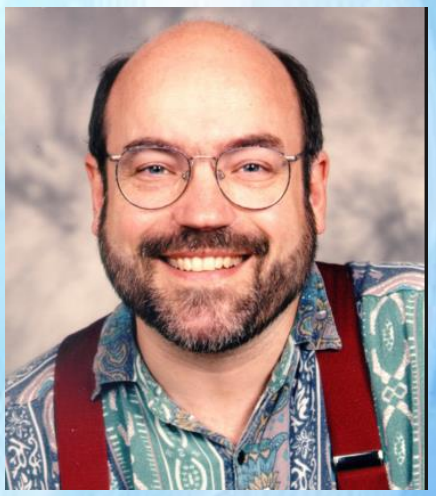
```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
6:          for(; i<100;i++)
7:              if (i%2 == 1)
9:              else:
10:                 z += 1;
11:         printf("%d\n",z);
13:     }
```

3、静态切片

- 基于数据流方程求解
  - 针对CFG进行数据流分析，获取其中每个节点的相关变量集合
    - 最初只获取后向静态切片
    - 针对拥有非结构化控制流的程序切片不够精确
- 基于图可达性算法
  - 将切片问题转化为程序依赖图的可达性问题

3、静态切片 – 数据流方程

- 数据流方程切片
  - –Mark Weiser博士在其1979年博士论文中建立
  - –1981年在国际软件工程会议上首次提出
  - –经典论文《Program Slicing》



普适计算之父

https://cse.buffalo.edu/LRG/CSE705/Papers/Weiser-Static-Slicing.pdf

3、静态切片 – 数据流方程

- 数据流方程求解基本思想
  - 计算程序控制依赖（CDG）
  - 两层多轮迭代
    - 1. 数据依赖回溯提取相关变量和语句
    - 2. 若第1步提取的变量所在语句控制依赖于某个变量，则利用控制依赖回溯提取间接相关变量和语句
    - 3. 基于新提取的变量和语句重复执行第1步，直至得到的所有相关语句的集合不再增大

```
1:    int main() {
2:      int x, y, z;
3:      int i=0;
4:      z = 0;
5:      y = getchar();
6:      for(; i<100;i++)
7:        if (i%2 == 1)
8:          x += y*i;
9:        else
10:         z += 1;
11:     printf("%d\n",x);
12:     printf("%d\n",z);
13:   }
```

3、静态切片 – 数据流方程

- 基本概念
  - –针对节点 i
    - **Def(i)，Ref(i)**：定义集，引用集
    - **Infl (i)** ：控制依赖于i的节点集合，或i所控制的节点
    - $\mathbf{R^k_C(i)}$：相关变量集合
  - –针对
    - $\mathbf{S^k_C}$：相关语句集合
    - $\mathbf{B^k_C}$：相关分支语句集合

3、静态切片 – 数据流方程

- 基本概念的计算公式
- $R^k_C(i)$
  - $k = 0$

公式1

$$\bigcup_{\forall v_i \to_{cfg} v_j} \{u \mid u \in Ref(v_i) \wedge ((Def(v_i) \cap R^0_c(j)) \neq \varnothing)\} \cup \{u \mid u \notin Def(v_i) \wedge u \in R^0_C(j)\}$$

公式2

  - $k>0$

$$R^{k+1}_C(i) = R^k_C \cup \bigcup_{b \in B^k_C} R^0_{(b,Ref(b))}(i)$$

公式3

- B^k_C

$$B^k_C = \{b \mid i \in Infl(b), i \in S^k_C\}$$

公式4

- S^k_C

$$S^0_C = \{i \mid Def(i) \cap R^0_C(j) \neq \varnothing, i \to_{cfg} j\}$$

公式5

$$S^{k+1}_C = B^k_C \cup \{i \mid Def(i) \cap R^{k+1}_C(i) \neq \varnothing, i \to_{cfg} j\}$$

3、静态切片 – 数据流方程

- 基本的数据依赖关系的计算:k=0，复杂度elog(e)
- $R^k_C(i)$： 针对每个节点都要计算的相关变量集合
  - k = 0

公式1

$$\bigcup_{\forall v_i \to_{cfg} v_j} \{u \mid \boxed{u \in Ref(v_i)} \wedge ((Def(v_i) \cap R^0_c(j)) \neq \varnothing)\} \cup \{u \mid u \notin Def(v_i) \wedge \boxed{u \in R^0_C(j)}\}$$

- 假设当前节点为V_u，其后继节点为V_m，计算$R^0_C(u)$，切片准则<n, V>
- 当V_u = n
  - (a) $R^0_C(u) = V \cup \{v \mid v \in Ref(u) \wedge (V \cap Def(V) \neq \emptyset)\}$
- 若V_u != n
  - (b) 传播：如果V_u是V_m中相关变量的一个定义，则把V_u中相应引用的变量添加到相关变量集合$R^0_C(V_u)$中
  - (c) 有条件继承：若某变量在V_m（ V_u的后继节点）中是相关的变量，并且该变量没有在V_u中定义，则在语句V_u中仍然是相关变量

3、静态切片 – 数据流方程

- 基本的数据依赖关系的计算
- $R^k_C(i)$： 针对每个节点都要计算的相关变量集合
  - $k = 0$

公式1

$$\bigcup_{\forall\, v_i \to_{cfg} v_j} \{u \mid u \in Ref(v_i) \wedge ((Def(v_i) \cap R^0_c(j)) \neq \varnothing)\} \cup \{u \mid u \notin Def(v_i) \wedge u \in R^0_C(j)\}$$

```
1:    y = x
2:    a = b
3:    z = y
```

$R^0_{<3,\{Y\}>}(1) = \{X\}$ 适用条件b

$R^0_{<3,\{Y\}>}(2) = \{Y\}$ 适用条件c

$R^0_{<3,\{Y\}>}(3) = \{Y\}$ 适用条件a

3、静态切片 – 数据流方程

- 控制依赖的计算
- B^k_C： 相关变量的控制依赖节点集合

公式3

$$B_C^k = \{b \mid i \in \mathrm{Infl}(b), i \in S_C^k\}$$

```
1:    i = 5;
2:    if ( i > 0 ) {
3:          a += 2;
4:          b  = a + 1;
      }
```

$$Infl(2) = \{3, 4\}$$

$$S^0_{<4,\{b\}>} = \{3, 4\}$$

$$B^0_{<4,\{b\}>} = \{2\}$$

3、静态切片 – 数据流方程

- 控制依赖的计算
- $R^k_C(i)$： 针对每个节点都要计算的相关变量集合
  - $k > 0$

公式2

$$R_C^{k+1}(i) = R_C^k \cup \bigcup_{b \in B_C^k} R^0_{(b,\mathrm{Ref}(b))}(i)$$

a) 继承上一次迭代的相关变量集合

b) 添加控制依赖节点引用变量的直接相关变量

```
1:    i = 5;
2:    if ( i > 0 ) {
3:        a += 2;
4:        b  = a + 1;
      }
```

$R^0_{<4,\{b\}>}(1)=\{a,b\}$

$R^0_{<4,\{b\}>}(2)=\{a,b\}$

$R^0_{<4,\{b\}>}(3)=\{a,b\}$

$R^0_{<4,\{b\}>}(4)=\{a,b\}$

$R^0_{<2,\{i\}>}(1)=\{\}$

$R^0_{<2,\{i\}>}(2)=\{i\}$

$R^0_{<2,\{i\}>}(3)=\{\}$

$R^0_{<2,\{i\}>}(4)=\{\}$

$R^1_{<4,\{b\}>}(1)=\{a,b\}$

$R^1_{<4,\{b\}>}(2)=\{a,b,i\}$

$R^1_{<4,\{b\}>}(3)=\{a,b\}$

$R^1_{<4,\{b\}>}(4)=\{b\}$

3、静态切片 – 数据流方程

TCA

- 基本概念的解释
- S^k_C：相关语句集合
  - k =0 $S_C^0 = \{i \mid Def(i) \cap R_C^0(j) \neq \varnothing, i \rightarrow_{cfg} j\}$ 公式4
  - k >0 $S_C^{k+1} = B_C^k \cup \{i \mid Def(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow_{cfg} j\}$ 公式5

a) 控制依赖节点（语句）

b) 是后继节点相关变量的定义的节点（语句）

```
1:    i = 5;
2:    if ( i > 0 ) {
3:        a += 2;
4:        b  = a + 1;
      }
```

$R^0_{<4,\{b\}>}(1)=\{a,b\}$

$R^0_{<4,\{b\}>}(2)=\{a,b\}$

$R^0_{<4,\{b\}>}(3)=\{a,b\}$

$R^0_{<4,\{b\}>}(4)=\{a,b\}$

$R^1_{<4,\{b\}>}(1)=\{a,b\}$

$R^1_{<4,\{b\}>}(2)=\{a,b,i\}$

$R^1_{<4,\{b\}>}(3)=\{a,b\}$

$R^1_{<4,\{b\}>}(4)=\{b\}$

$S^0_{<4,\{b\}>} = \{3,4\}$

$S^1_{<4,\{b\}>} = \{1,2,3,4\}$

3、静态切片

TCA

算法：Mark Weiser数据流切片算法

- **输入**：CFG G = { V,E, Entry,Exit }
- **输出**：Slice

① 第一次计算，按照公式1和公式2计算直接相关变量和语句,得到$\mathbf{R^0_C}$和$\mathbf{S^0_C}$。

② 进入循环

a) 依据公式3、4计算间接相关变量和间接相关语句，得到$\mathbf{R^k_C}$和$\mathbf{S^k_C}$

b) 依据公式5把控制节点添加到间接相关语句中，同时重新计算间接相关语句

c) 如果$\mathbf{S^k_C}$增大，则重复执行第2步

3、静态切片

- 数据流方程求解

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
4:          z = 0;
5:          y = getchar();
6:          for(; i<100;i++)
7:              if (i%2 == 1)
8:                  x += y*i;
9:              else
10:                 z += 1;
11:         printf("%d\n",x);
12:         printf("%d\n",z);
13:     }
```

切片准则 （12 , z)

第一轮（数据依赖）：
相关变量集合：{z}
切片语句集合：{12,10,4}

第二轮（第一次引入控制依赖）：
相关变量集合：{z, i}
切片语句集合：{12,10,7,6,4,3}

第三轮（控制依赖+数据依赖）：
相关变量集合：{z, i}
切片语句集合：{12,10,7,6,4,3}

切片：
{12,10,7,6,4,3}

3、静态切片

- 数据流方程求解

```
1:  int main() {
2:    int x, y, z;
3:    int i=0;
4:    z = 0;
5:    y = getchar();
6:    for(; i<100;i++)
7:      if (i%2 == 1)
8:          x += y*i;
9:      else
10:         z += 1;
11:   printf("%d\n",x);
12:   printf("%d\n",z);
13: }
```

第1轮

语句编号	12	11	10	8	7	6	5	4	3	2
R^0_C	z	z	z	z	z	z	z			
S^0_C	4, 10,12									

第2轮

B^0_C	6,7									
$R^0_{C(7,\{i\})}$			i	i	i	i	i	i		
$R^0_{C(6,\{i\})}$			i	i	i	i	i	i		
R^1_C	z	z	z,i	z,i	z,i	z,i	z,i	z,i	z	z
S^1_C	3, 4, 6, 7, 10,12									

第3轮

B^1_C	6,7									
$R^0_{C(7,\{i\})}$			i	i	i	i	i	i		
$R^0_{C(6,\{i\})}$			i	i	i	i	i	i		
R^2_C	z	z	z,i	z,i	z,i	z,i	z,i	z,i	z	z
S^2_C	3, 4, 6, 7, 10,12									

3、静态切片

- 数据流方程求解

```
1:  int main() {
2:    int x, y, z;
3:    int i=0;
4:    z = 0;
5:    y = getchar();
6:    for(; i<100;i++)
7:      if (i%2 == 1)
8:          x += y*i;
9:      else
10:         z += y;
11:   printf("%d\n",x);
12:   printf("%d\n",z);
13: }
```

第1轮

语句编号	12	11	10	8	7	6	5	4	3	2
R^0_C										
S^0_C										

第2轮

B^0_C										
$R^0_{C(7,\{i\})}$										
$R^0_{C(6,\{i\})}$										
R^1_C										
S^1_C										

第3轮

B^1_C										
$R^0_{C(7,\{i\})}$										
$R^0_{C(6,\{i\})}$										
R^2_C										
S^2_C										

3、静态切片

- 数据流方程求解

```
1:  int main() {
2:    int x, y, z;
3:    int i=0;
4:    z = 0;
5:    y = getchar();
6:    for(; i<100;i++)
7:      if (i%2 == 1)
8:        x += y*i;
9:      else
10:       z += y;
11:   printf("%d\n",x);
12:   printf("%d\n",z);
13: }
```

第1轮

语句编号	12	11	10	8	7	6	5	4	3	2
R^0_C	z	z	z,y	z,y	z,y	z,y	z			
S^0_C	4, 5, 10,12									

第2轮

B^0_C	6,7									
$R^0_{C(7,\{i\})}$			i	i	i	i	i	i		
$R^0_{C(6,\{i\})}$			i	i	i	i	i	i		
R^1_C	z	z	z,y,i	z,y,i	z,y,i	z,y,i	z,i	z,i	z	z
S^1_C	3, 4, 5, 6, 7, 10,12									

第3轮

B^1_C	6,7									
$R^0_{C(7,\{i\})}$			i	i	i	i	i	i		
$R^0_{C(6,\{i\})}$			i	i	i	i	i	i		
R^2_C	z	z	z,y,i	z,y,i	z,y,i	z,y,i	z,i	z,i	z	z
S^2_C	3, 4, 5, 6, 7, 10,12									

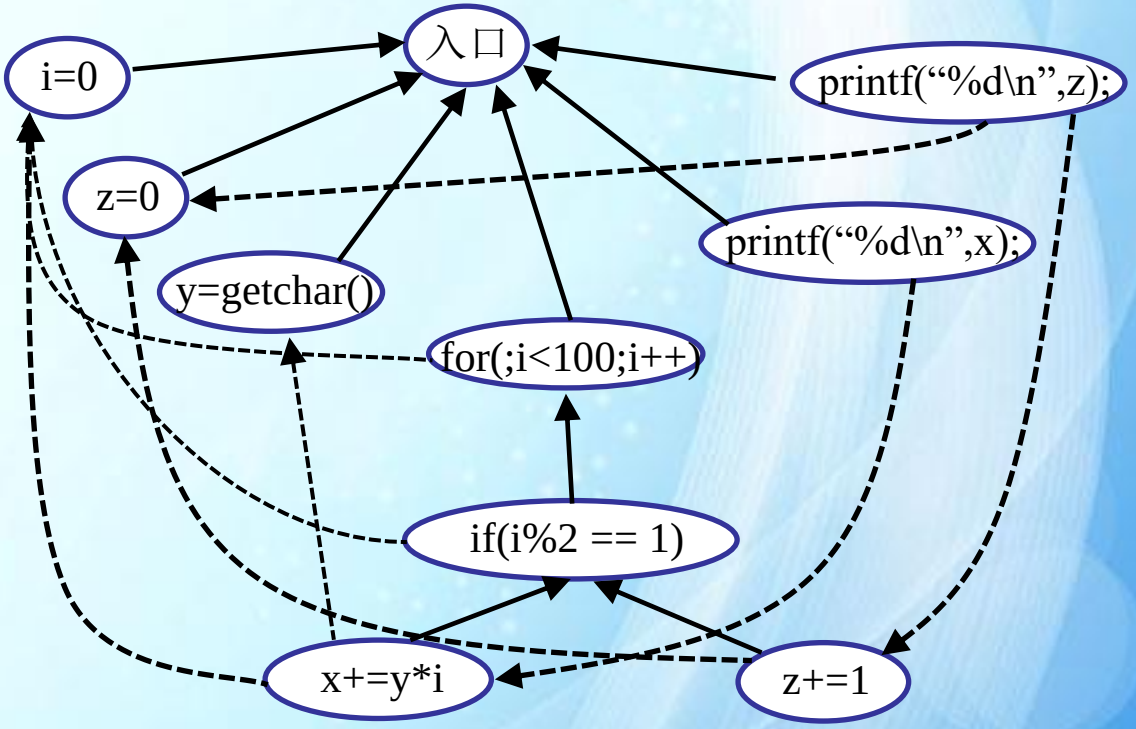
TCA

- 基于数据流方程求解 – 小结
  - 方法
    - 基于数据依赖和控制依赖关系进行迭代求解
  - 局限性
    - 理解困难：
      - 涉及的概念较多，计算过程比较复杂，不够直观
  - 支持多个过程/函数的数据流方程切片算法？

3、静态切片

- 基于图可达性的切片计算

```
1:      int main() {
2:          int x, y, z;
3:          int i=0;
4:          z = 0;
5:          y = getchar();
6:          for(; i<100;i++)
7:              if (i%2 == 1)
8:                  x += y*i;
9:              else
10:                 z += 1;
11:         printf("%d\n",x);
12:         printf("%d\n",z);
13:     }
```



3、静态切片

- 基于图可达性的切片求解
  - 构造目标程序的程序依赖图（PDG)
  - 制定程序切片规则C
  - 依据切片规则，在PDG上应用图可达性算法，获得与切片相对应的依赖图
  - 将依赖图切片映射到目标程序中，得到依赖图对应的语句集合，该语句集合形成目标程序基于切片准则C的程序切片

- 基于图可达性的静态切片算法

算法：Slice（节点n)：基于图遍历递归的切片

1. 标记当前节点n
2. 对于节点n的每个前必经节点m

    若m没有被标记

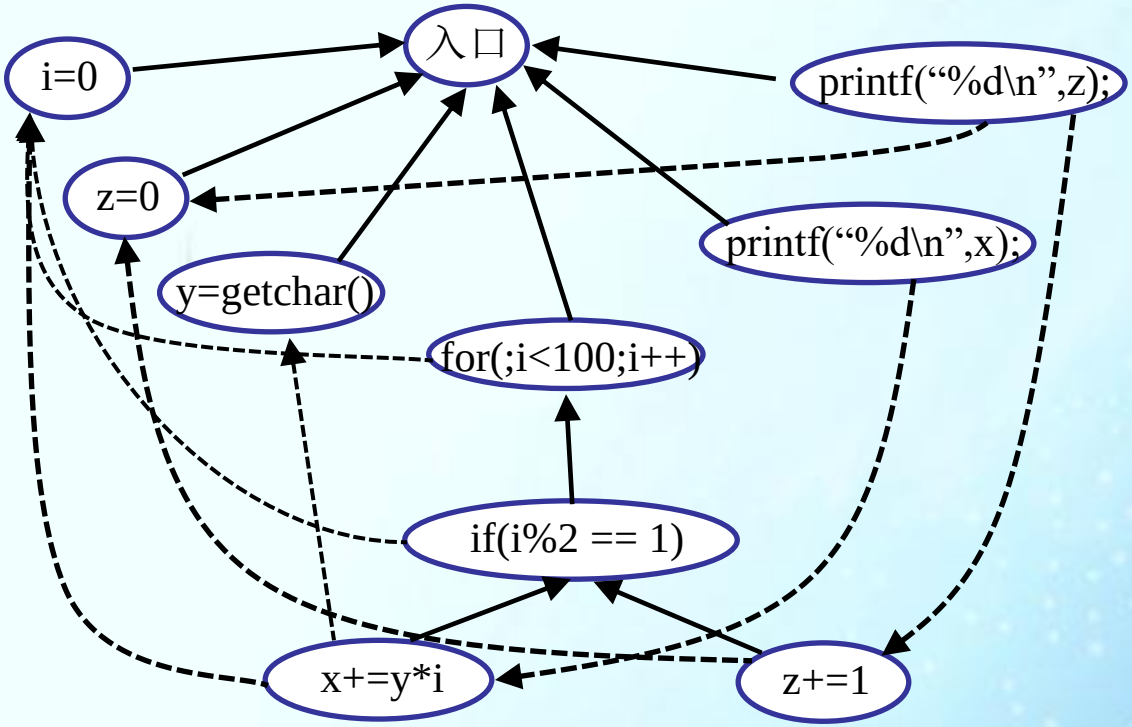
        标记节点m

        对于所有m所依赖的前必经节点 f

            Slice(f)

3、静态切片

- 基于图可达性的静态切片算法示例



切片准则 （12 , z)

切片
{12,10,7,6,4,3}

TCA

静态切片小结

- 方法
  - 数据流方程
    - 数据依赖和控制依赖关系的迭代求解计算，计算过程复杂
  - 图可达性算法
    - 直观，计算简便，实用性较强
- 局限
  - 路径爆炸
    - 针对规模较大程序，切片过大

4、动态切片

- 动态切片
  - 由学者B.Korel和J.Laski最早于1988年提出
    - 基于数据流方程的动态切片计算
    - 经典文献《Debugging with dynamic slicing and backtracking》
  - H.Agrawal和J.Horgan于1990年提出基于图的动态切片计算方法
    - 经典文献《Dynamic Program Slicing》

http://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/slicing.pdf

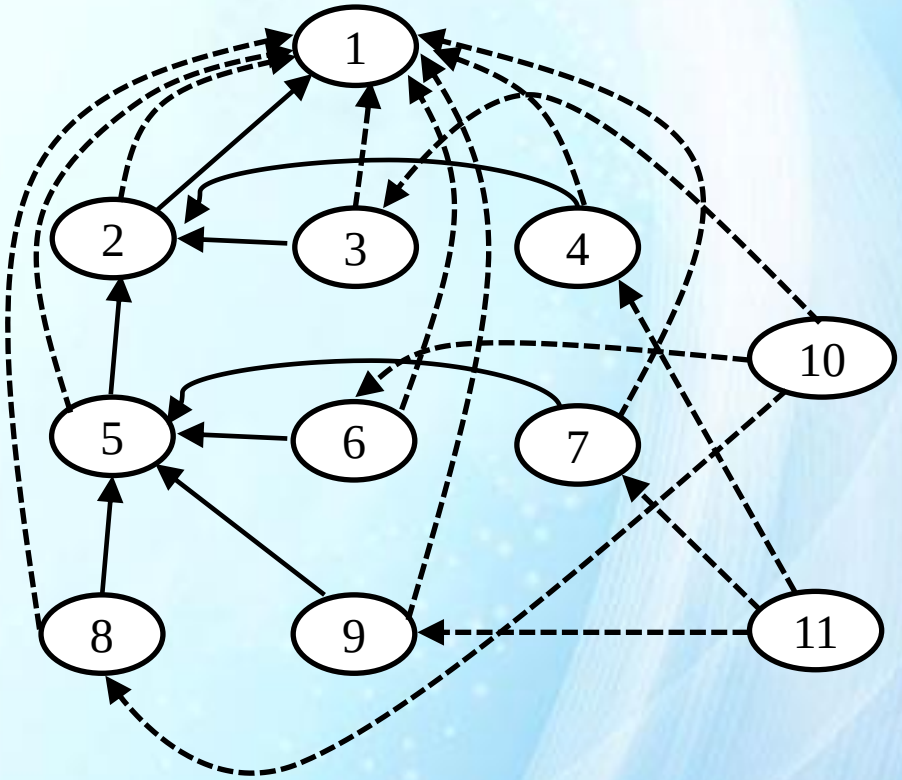
4、动态切片

- 与特定输入相关的程序切片
  - 给定输入情况下，程序执行路径上所有对程序中某个点上的变量的值有影响的语句集合
  - 切片准则$<n, V, I_0>$
- 特点
  - 相比于静态切片，精确度更高，执行代价较高

4、动态切片

示例（切片准则 < 10 , y >）

```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:          y = f3(x);
9:          z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```



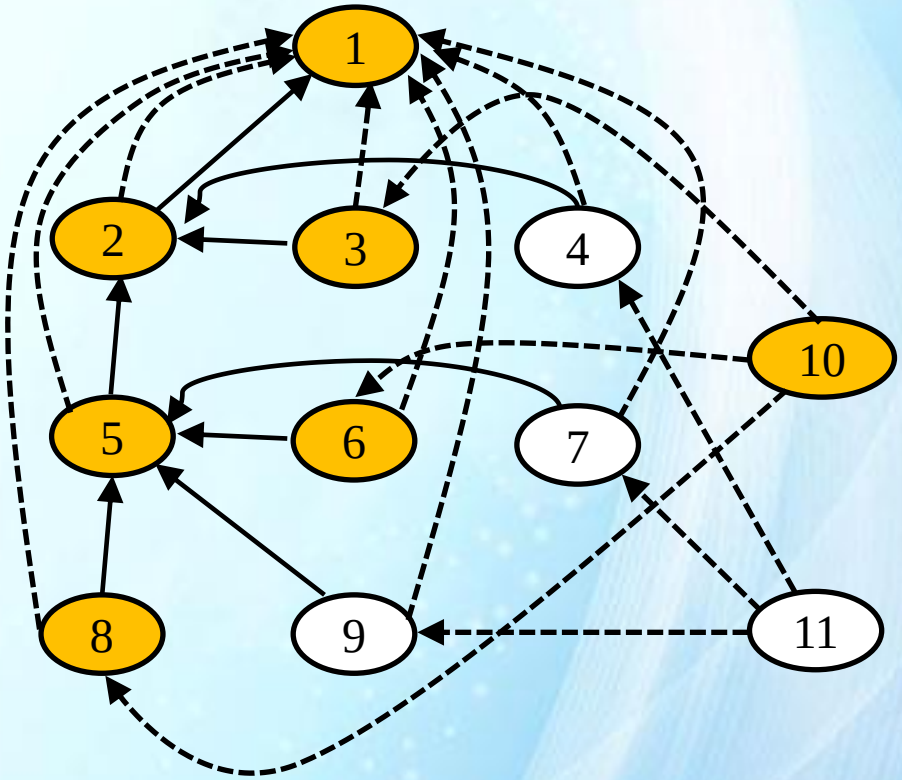
静态切片

{1，2，3，5，6，8，10}

4、动态切片

示例（切片准则 < 10 , y >）

```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:            y = f3(x);
9:            z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```



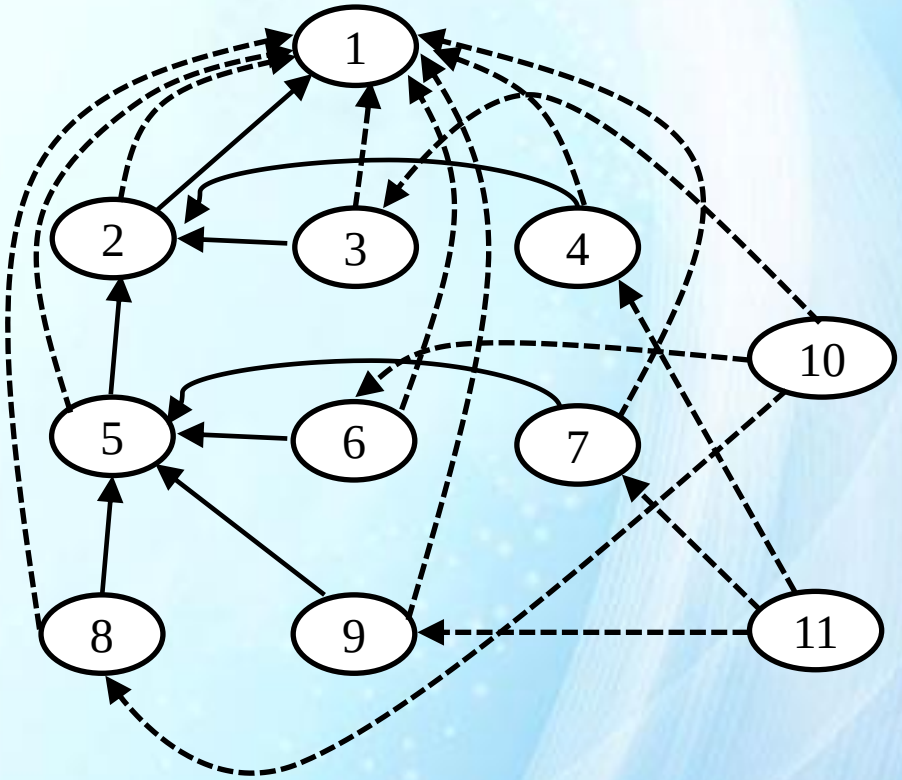
静态切片

{1，2，3，5，6，8，10}

4、动态切片

示例（切片准则 < 10 , y >）输入 X = -1

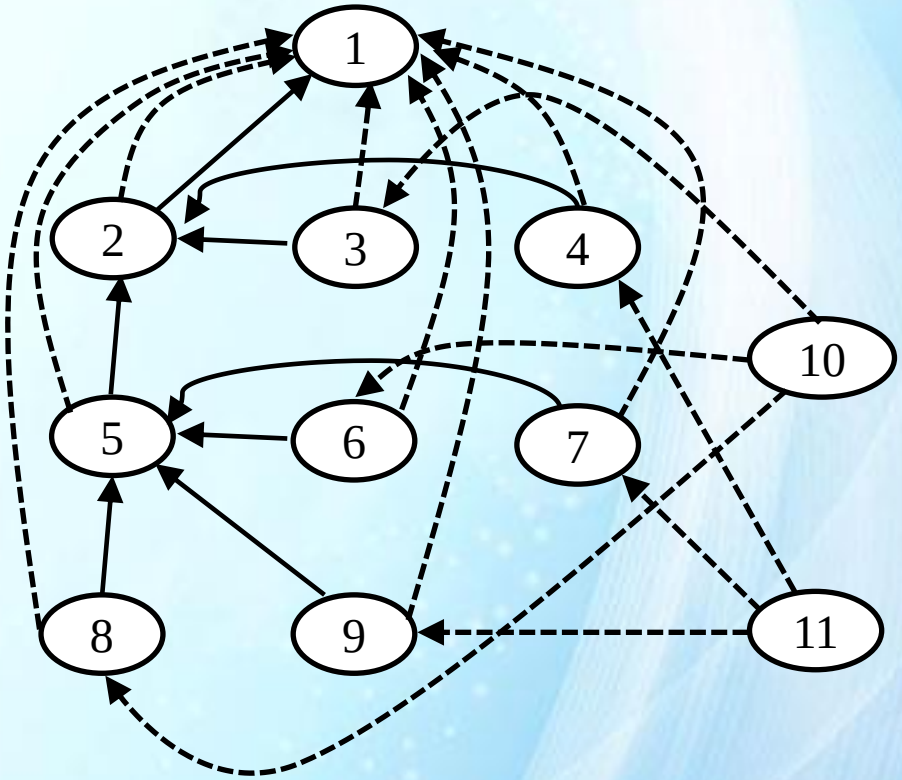
```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:          y = f3(x);
9:          z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```



4、动态切片

示例（切片准则 < 10 , y >）输入 X = -1

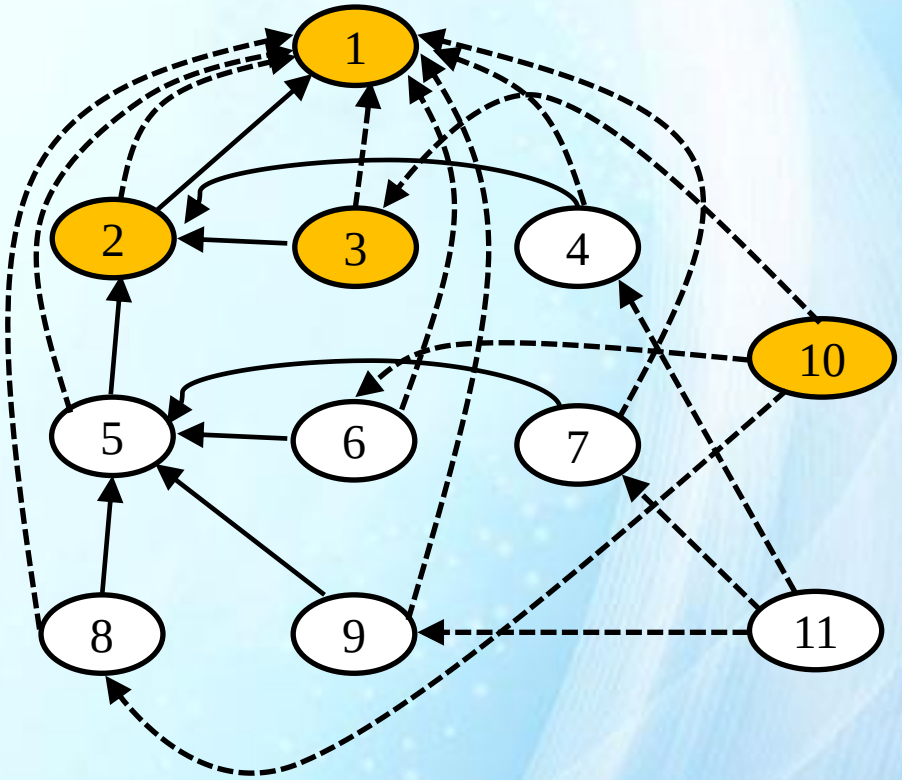
```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:          y = f3(x);
9:          z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```



4、动态切片

示例（切片准则 < 10 , y >）输入 X = -1

```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:            y = f3(x);
9:            z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```



切片
{1，2，3，10}

4、动态切片

- 执行历史(Trace)
  - 程序控制流图路径
  - 由遍历控制流图得到的节点构成
- 切片准则
  - <n, V, I_0>
  - I_0: 程序特定输入

- 输入 X = -1

```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:          y = f3(x);
9:          z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```

<1, 2, 3, 10, 11>

- 输入 N = 2

```
1:      N = getchar();
2:      z = 0;
3:      y = 0;
4:      i = 1;
5:      while(i <= N){
6:          z = f1(z,y);
7:          y = f2(y);
8:          I = I + 1
        }}
9:      printf("%d\n",z);
```

<1, 2, 3, 4, 5^1, 6^1, 7^1, 8^1, 5^2, 6^2, 7^2, 8^2, 5^3, 9>

4、动态切片

TCA

基于程序依赖图的切片方法（方法一）

1. 程序依赖图 G = (V,E)，切片准则节点为n
2. 依据程序输入获取程序执行历史H
3. 依据H对G中节点进行标记

```
for u in V:
    If u 属于 H:
        u.marked = true
```

4. 删除G中未标记节点

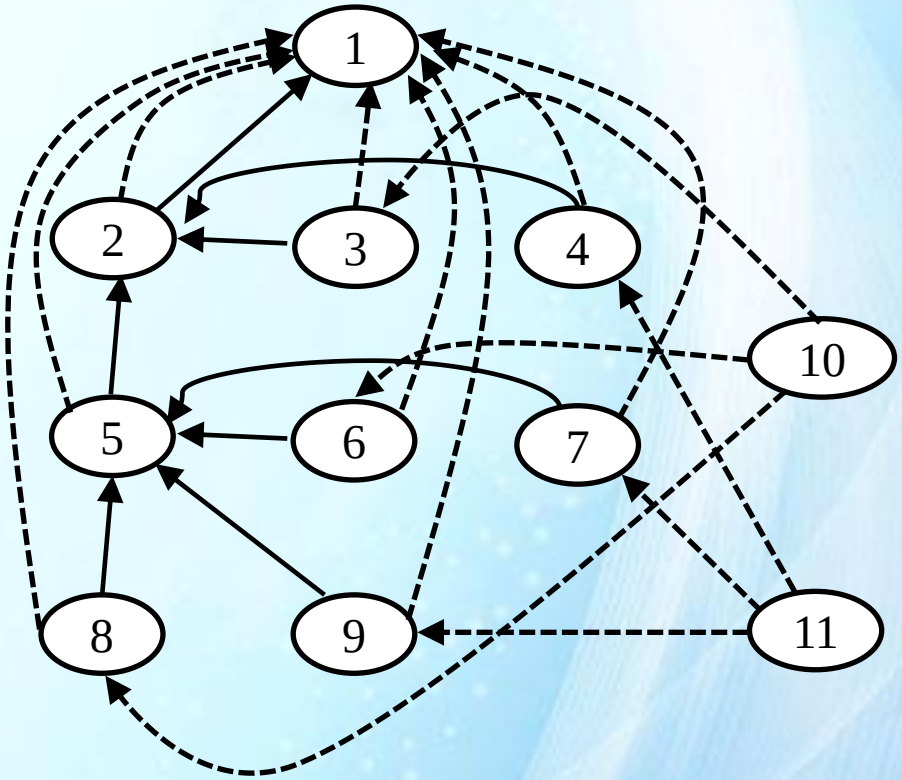
```
for u in V:
    if u.marked = false
        G.remove(u)
```

5. 从节点n开始对G进行遍历，得到切片

4、动态切片

- 方法一示例（切片准则 < 10 , y >） 输入 X = -1

```
        int x, y, z;
1:      x = getchar()
2:      if (x < 0){
3:          y = f1(x);
4:          z = g1(x);
        }else{
5:        if (x == 0){
6:            y = f2(x);
7:            z = g2(x);
        }else{
8:          y = f3(x);
9:          z = g3(x);
        }}
10:     printf("%d\n",y);
11:     printf("%d\n",z);
```



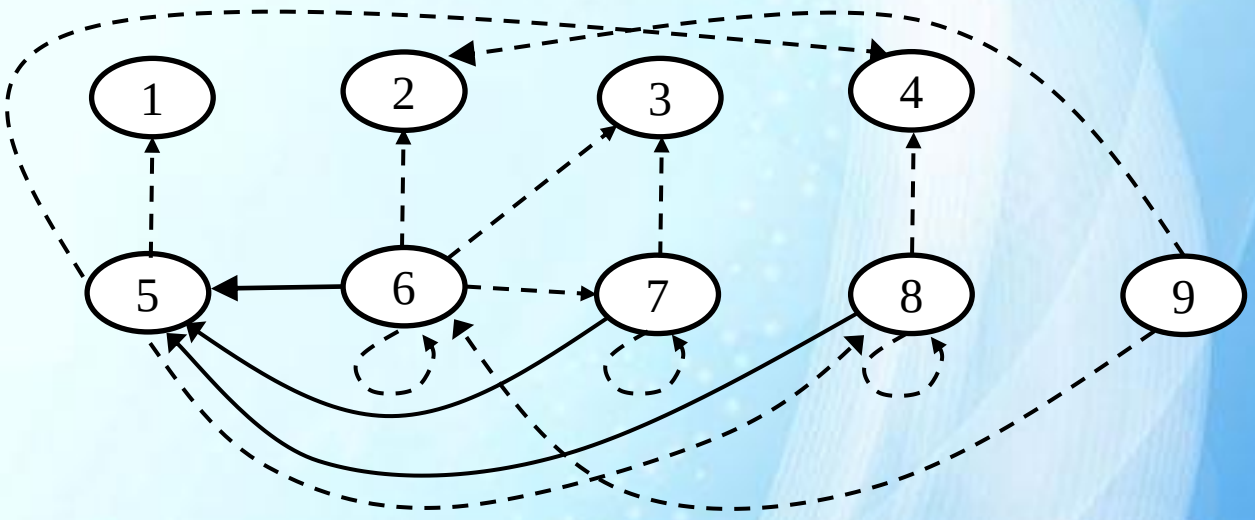
切片
{1，2，3，10}

4、动态切片

- 基于程序依赖图的切片方法的不足
  - 切片准则 < 9 , {z} >）　输入 N = 1

执行历史：　<1，2，3，4，5^1，6，7，8，5^2，9>

```
1:      N = getchar();
2:      z = 0;
3:      y = 0;
4:      i = 1;
5:      while(i <= N){
6:          z = f1(z,y);
7:          y = f2(y);
8:          I = I + 1
        }}
9:      printf("%d\n",z);
```



切片

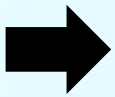
{{1，2，3，4，5，6，7，8，9}

4、动态切片

- 基于程序依赖图的切片方法的不足
  –切片准则 < 9 , {z} >）　输入 N = 1

  执行历史：　<1，2，3，4，5^1，6，7，8，5^2，9>

```
1:      N = getchar();
2:      z = 0;
3:      y = 0;
4:      i = 1;
5:      while(i <= N){
6:              z = f1(z,y);
7:              y = f2(y);
8:              i = i + 1
                }}
9:      printf("%d\n",z);
```



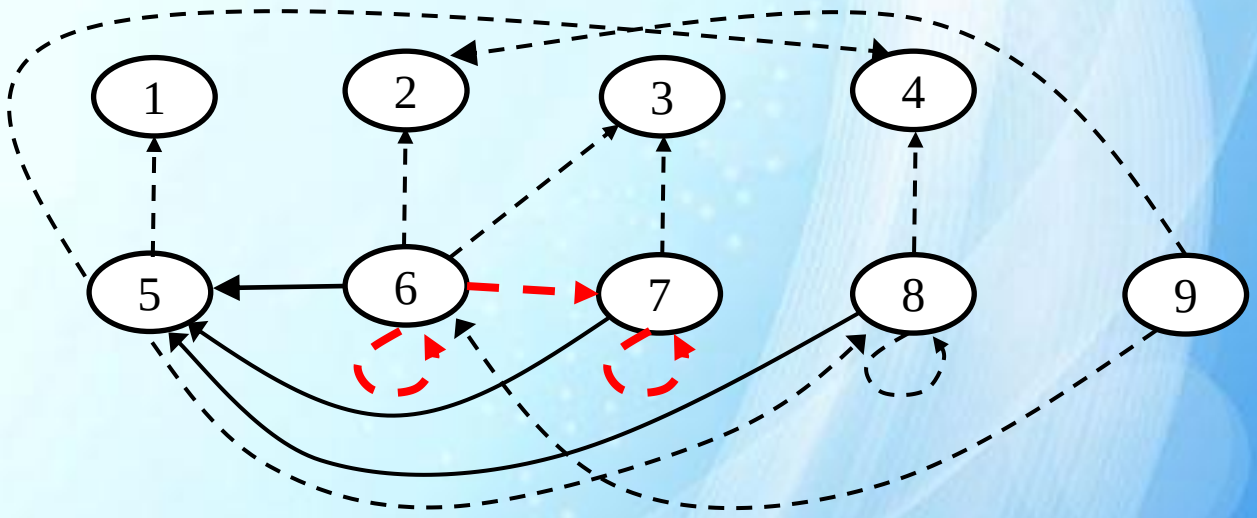
```
1:      N = getchar();
2:      z = 0;
3:      y = 0;
4:      i = 1;
5:      while(i <= N){
6:              z = f1(z,y);
7:              y = f2(y);
8:              i = i + 1
                }}
9:      printf("%d\n",z);
```

4、动态切片

- 基于程序依赖图的切片方法的不足
  - –执行历史包含的节点与切片准则不相关
    - 任一执行路径上的任意一条语句最多只能包含某个变量的一个可到达定义

```
1:      N = getchar();
2:      z = 0;
3:      y = 0;
4:      i = 1;
5:      while(i <= N){
6:          z = f1(z,y);
7:          y = f2(y);
8:          I = I + 1
        }}
9:      printf("%d\n",z);
```

执行历史： <1，2，3，4，5^1，6，7，8，5^2，9>



切片

{{1，2，3，4，5，6，8，9}

4、动态切片

基于程序依赖图的切片方法（方法二）

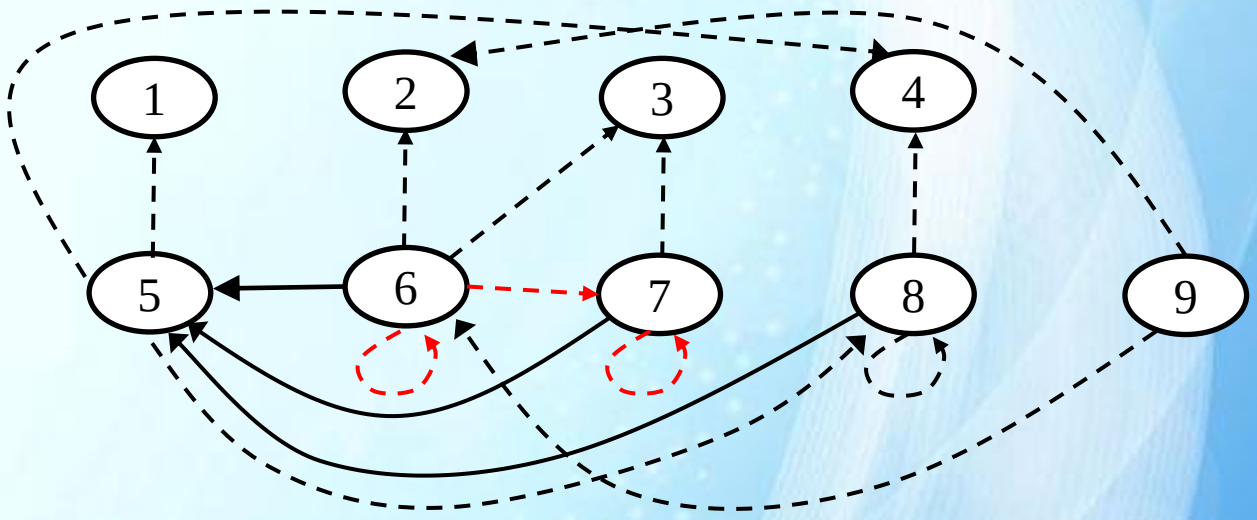
1. 程序依赖图 G = (V,E)，切片准则节点为n
2. 依据程序输入获取程序执行历史H
3. 依据H对G中节点和边进行标记
4. 删除G中未标记节点和未标记边
5. 从节点n开始对G进行遍历，得到切片

4、动态切片

- 方法二示例

执行历史： <1，2，3，4，5^1，6，7，8，5^2，9>

```
1:      N = getchar();
2:      z = 0;
3:      y = 0;
4:      i = 1;
5:      while(i <= N){
6:          z = f1(z,y);
7:          y = f2(y);
8:          I = I + 1
        }}
9:      printf("%d\n",z);
```



切片

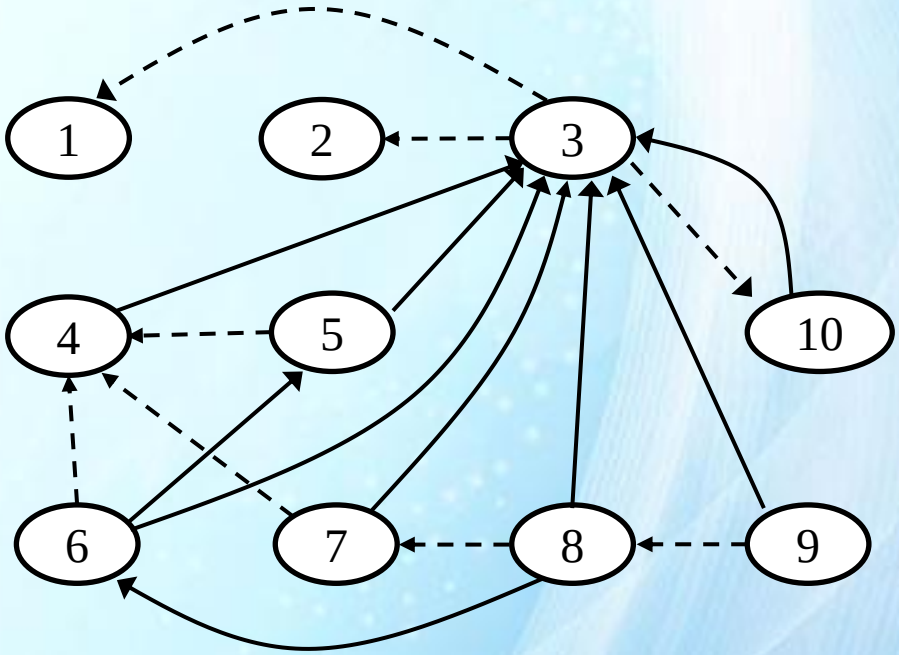
{{1，2，3，4，5，6，8，9}

4、动态切片

- 切片准则 < 9^2 , z > 输入 N = 2, x分别为1和-1

执行历史： <1，2，3^1，4^1，5^1，6，8^1，9^1，10^1，3^2，4^2，5^2，7，8^2，9^2>

```
1:      N = getchar();
2:      I = 1;
3:      while(I <= N){
4:          x = getchar();
5:          if (x < 0)
6:              y = f1(x);
            else
7:              y = f2(x);
8:          z = f3(y);
9:          printf("%d\n",z);
10:         I = I + 1;
        }
```



4、动态切片

- 方法二的不足
  - 任意一条语句在执行历史中可能出现多次，每次执行可能使用某个变量的不同的可到达定义
  - 程序依赖图是一种静态表示，无法体现同一个语句（节点）的不同次执行的区别

4、动态切片

TCA

基于动态依赖图的切片（方法三）

1. 程序依赖图 G = (V,E)，切片准则节点为n
2. 依据程序输入获取程序执行历史H
3. 依据H创建动态依赖图P

for u in H:

创建节点u^i，i为节点出现递增序号

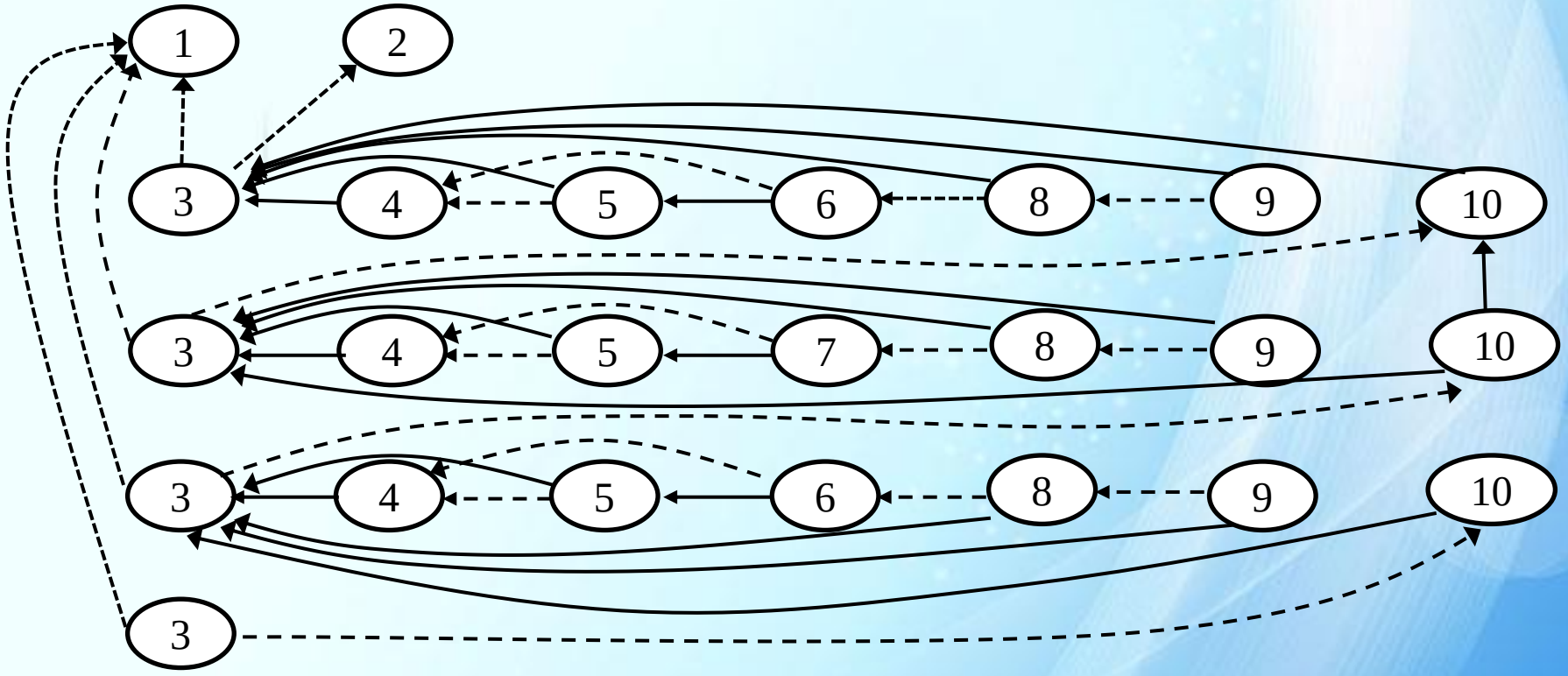
建立节点u^i与其执行时对应的控制依赖节点和数据依赖节点之间的边

4. 从最后一次执行的节点n开始对G进行遍历，得到切片

- 方法三示例

切片准则 < 9^3 , z > 输入 N = 3, **x分别为-4，3和-2**

执行历史： **<1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>**

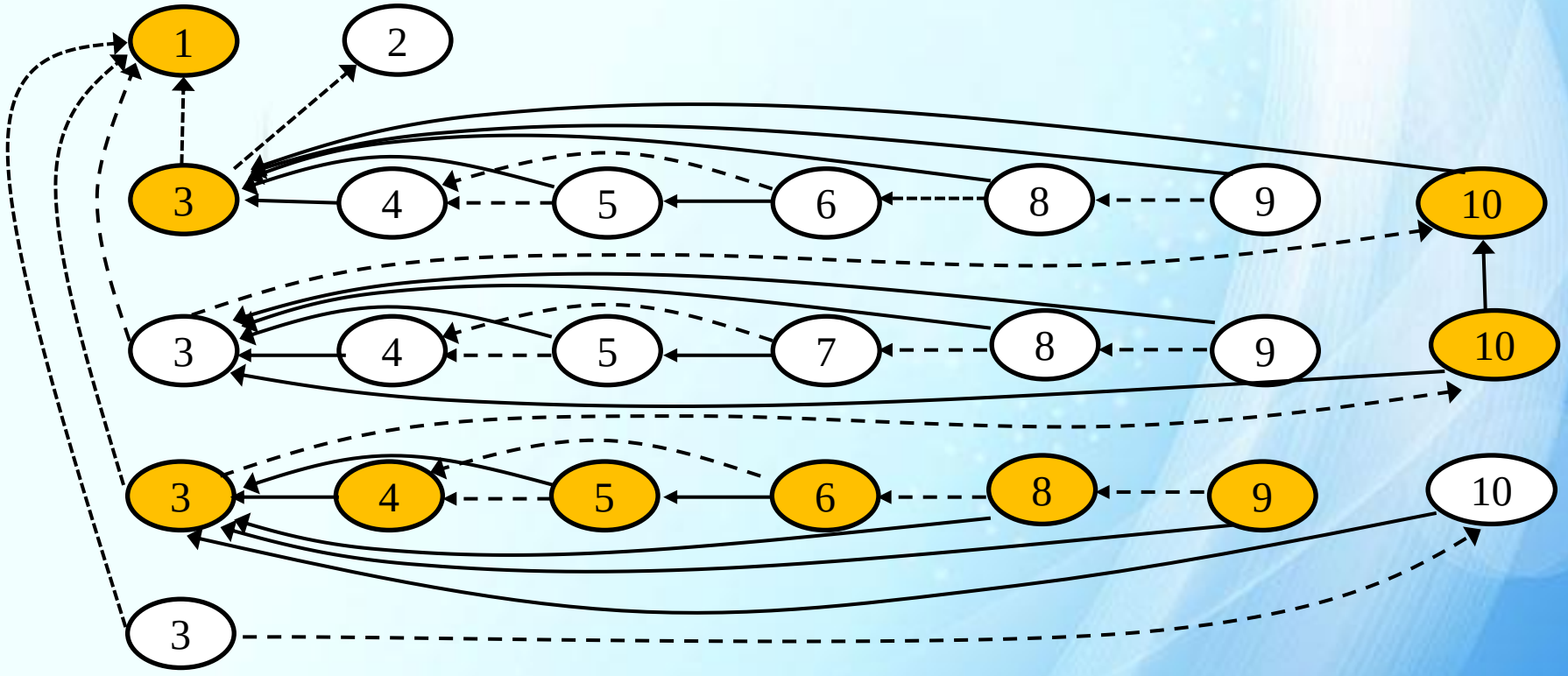


4、动态切片

- 方法三示例

切片准则 < 9^3 , z > 输入 N = 3, **x分别为-4，3和-2**

执行历史： **<1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>**

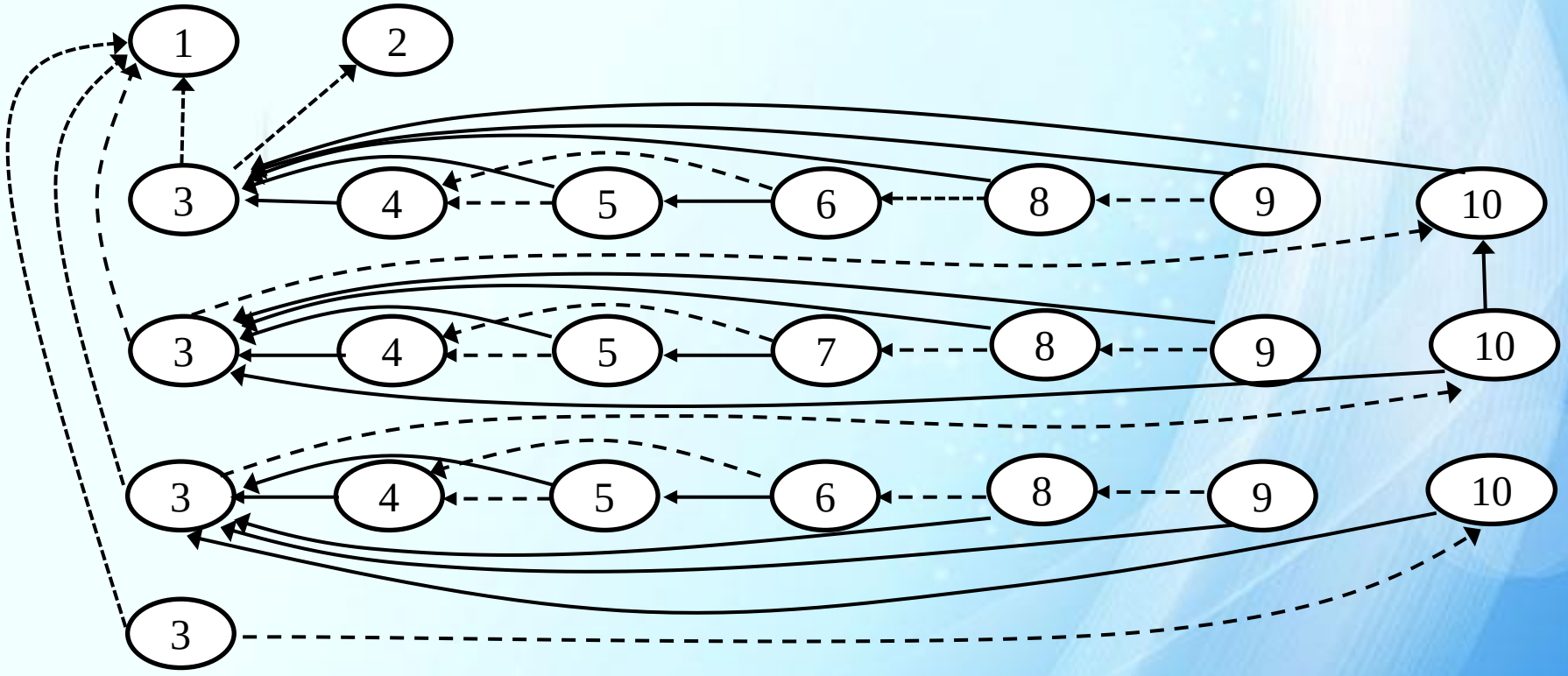


4、动态切片

- 方法三局限性
  - 节点数量与执行历史中的语句数量相同，而执行历史的长度与程序运行时的具体输入有关
    - 执行历史的长度没有限制（可能是无限大），动态依赖图的大小也因而没有上界
  - 由于程序规模限制，切片的规模不会很大，可能的切片数量也不会很大
    - 针对前述代码片段，如果N=10000，那么动态依赖图的节点数量将超过70000个，但是最终获得的动态切片的节点数量不会超过10个

4、动态切片

- 方法三的改进
  - –动态依赖图是否能化简
    - 能否将等价节点合并？ 什么是等价？节点序号，边



4、动态切片

- 动态依赖图化简
  - 合并编号相同且出边相同的节点
    - 对于两个出边相同的节点，以其为切片准则得到的切片相同
    - 出边包括直接数据依赖边和直接控制依赖边
  - 合并方法
    - 不能基于原始动态依赖图直接合并
    - 需要从头构建
      - 逐步合并等价节点

4、动态切片

简化动态依赖图构建方法

1. 依据程序输入获取程序执行历史H
2. 依据H创建动态依赖图P

for u in H:

若P为空

创建节点u^i，i为节点出现递增序号

若P不为空

查询当前依赖图，查看是否有节点v，使得$D_u \cup C_u = D_v \cup C_v$

存在v

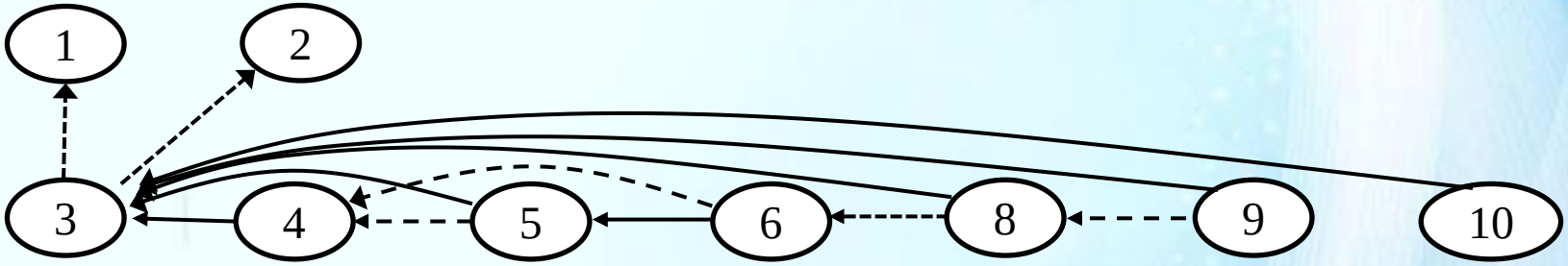
添加u到v中

不存在v

创建节点 u^i，从u^i向D_u 和C_u 中每个节点引出一条边

4、动态切片

- 相关概念的定义
  1. D_u ： 节点u的运行时直接数据依赖边
  2. C_u ： 节点u的运行时直接控制依赖边



$D_3 = \{1,2\}$ $D_5 = \{4\}$

$C_3 = \{\ \}$ $C_5 = \{3\}$

4、动态切片

- 相关概念的定义

1. DefineNode：当前节点引用的每个变量与当前图中最后一次定义该变量的节点编号的映射

   对应于D集合，直接数据依赖边

2. ControlNode ：当前节点的控制依赖条件节点与其在图中最近一次出现的节点的映射

   对应于C集合，直接控制依赖边

3. ReachableStmts：图中每个存在节点与其可到达的所有节点的集合

4、动态切片

- 切片计算示例

输入 N = 3, x分别为-4，3和-2

执行历史： <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>

```
1:      N = getchar();
2:      I = 1;
3:      while(I <= N){
4:          x = getchar();
5:          if (x < 0)
6:              y = f1(x);
            else
7:              y = f2(x);
8:          z = f3(y);
9:          printf("%d\n",z);
10:         I = I + 1;
        }
```

序号	D	C	R	新节点	新建边
1			1 -> 1	1	
2			2 -> 2	2	
3^1	N -> 1 I -> 2		3 -> 1,2,3	3	3 –> 1 3 -> 2
4^1		3 -> 3	4_1 -> 1,2,3,4_1	4_1	4_1 -> 3
5^1	X -> 4_1	3 -> 3	5_1 -> 1,2,3,4_1,5_1	5_1	5_1 -> 3 5_1 -> 4_1
6^1	X -> 4_1	5-> 5_1	6_1 -> 1,2,3,4_1,5_1,6_1	6_1	6_1 –> 5_1 6_1 –> 4_1
8^1	Y -> 6_1	3 -> 3	8_1 -> 1,2,3,4_1,5_1,6_1,8_1	8_1	8_1 -> 3,6_1
9^1	Z -> 8_1	3 -> 3	9_1 -> 1,2,3,4_1,5_1,6_1,8_1,9_1	9_1	9_1 -> 3,8_1

4、动态切片

- 切片计算示例

输入 N = 3, x分别为-4，3和-2

执行历史： <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>

```
1:      N = getchar();
2:      I = 1;
3:      while(I <= N){
4:          x = getchar();
5:          if (x < 0)
6:              y = f1(x);
            else
7:              y = f2(x);
8:          z = f3(y);
9:          printf("%d\n",z);
10:         I = I + 1;
        }
```

序号	D	C	R	新节点	新建边
10^1	I -> 2	3 -> 3	10->1,2,3	10	10 -> 2,3
3^2	I -> 10 N -> 1		循环依赖		3_2 -> 1,10

4、动态切片

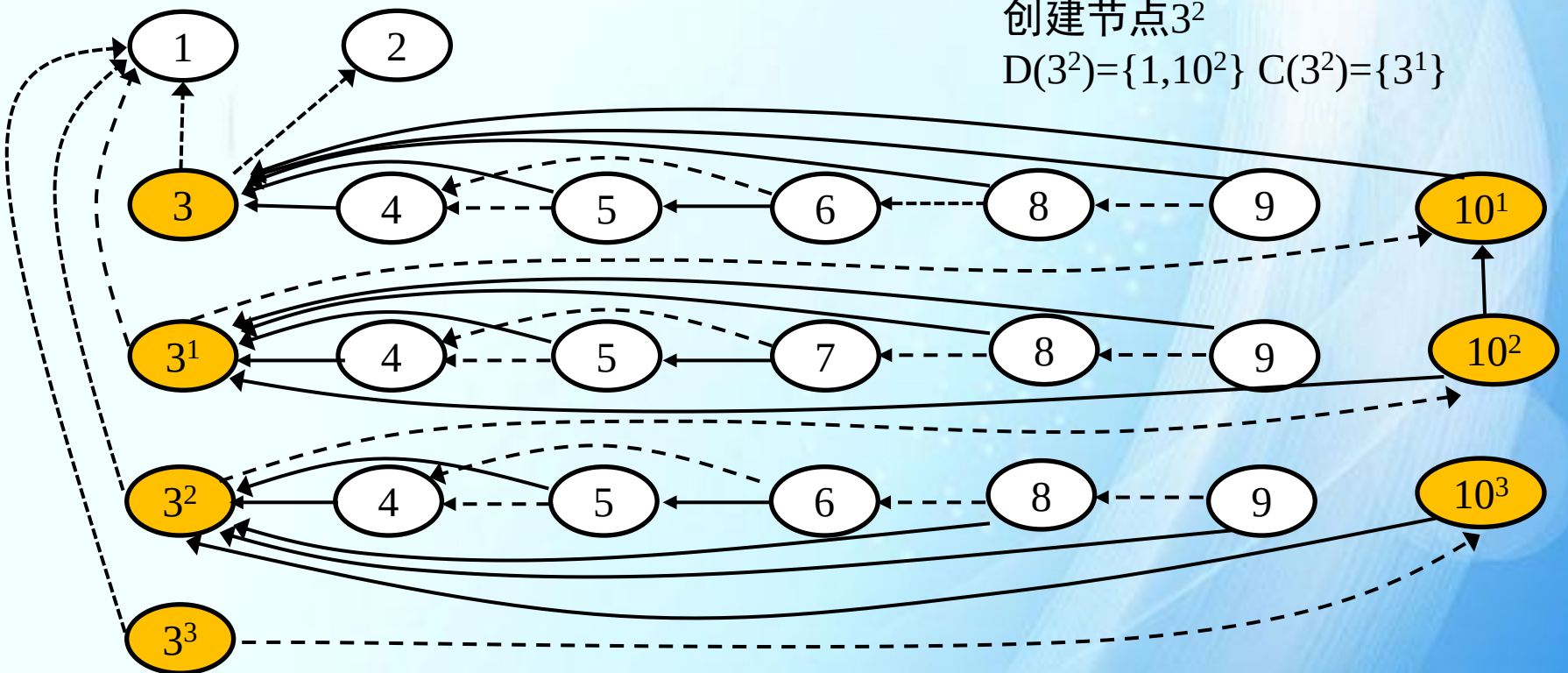
- 动态依赖图化简
  - 循环依赖问题
    - 节点3与节点10之间存在循环依赖，按照上述方法将不断产生节点3和10的节点

D(3)={1,2} C{3} = {}

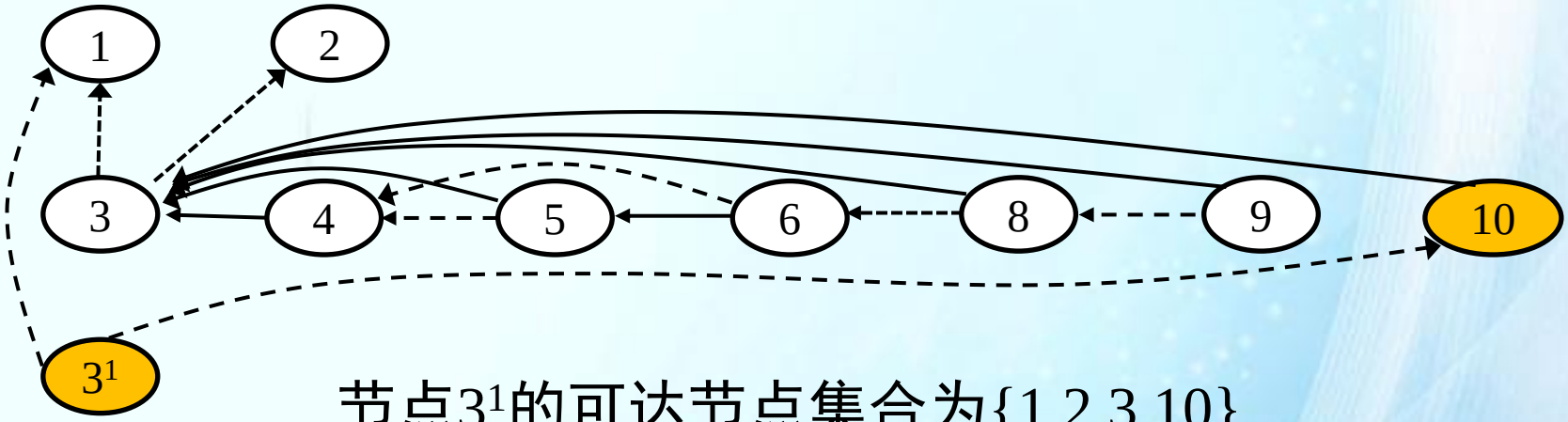
创建节点3^1
D(3^1)={1,10^1} C{3^1}={3}.

创建节点10^2
C（10^2）={3^1} D(10^2)={10^1}

创建节点3^2
D(3^2)={1,10^2} C(3^2)={3^1}



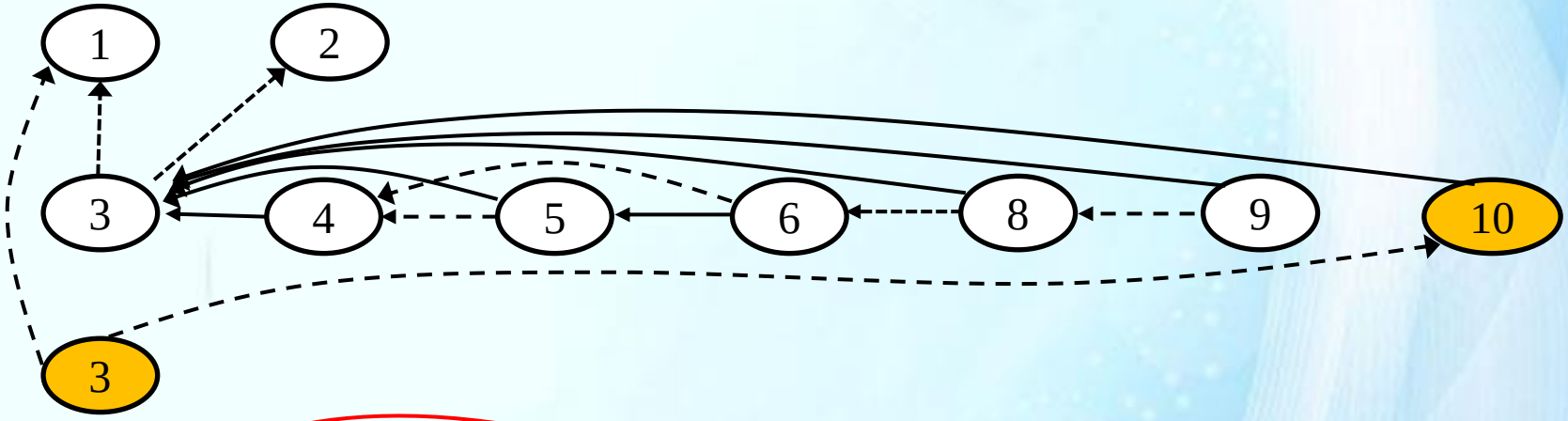
- 动态依赖图化简
  - –针对循环依赖的改进
    - 进一步判断当前要添加节点的可达节点集合是否是其某个直接后继节点v的可达节点集合的子集，如果是，则将不添加该节点，将其加入节点v中。

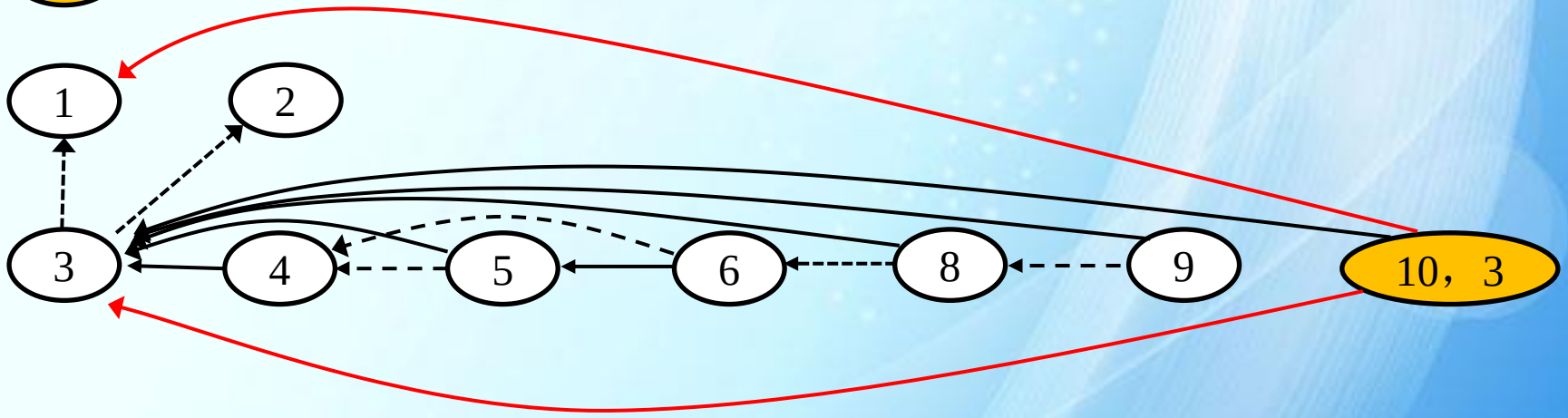


节点3^1的可达节点集合为{1,2,3,10}
节点3^1的后继节点10的可达节点集合为{1,2,3,10}
节点3^1的后继节点1的可达节点集合为{1}
节点3^1要合并到节点10中

4、动态切片

- 动态依赖图化简
  - –针对循环依赖的改进
    - 节点3与节点10合并





4、动态切片

- 切片计算示例

执行历史： <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>

```
1:      N = getchar();
2:      I = 1;
3:      while(I <= N){
4:          x = getchar();
5:          if (x < 0)
6:              y = f1(x);
            else
7:              y = f2(x);
8:          z = f3(y);
9:          printf("%d\n",z);
10:         I = I + 1;
        }
```

序号	D	C	R	新节点	新建边
10^1	I -> 2	3 -> 3	10->1,2,3	10	10 -> 2,3
3^2	I -> 10 N -> 1		循环依赖	替换为（10，3）	3_2 -> 1,10
4^2		3 ->(10，3)	4_2 -> 1,2,3,4_2,(10,3)	4_2	4_2 -> (10,3)
5^2	X->4_2	5->(10,3)	5_2 -> 1,2,3,(10,3),4_2,5_2	5_2	5_2 -> 4_2 5_2 -> (10,3)
7^1	X->4_2	5->5_2	7_1 -> 1,2,3,(10,3),4_2,5_2,7_1	7_1	7_1 -> 5_2 7_1 -> 4_2
8^2	Y -> 7_1	3 ->(10，3)	8_2 -> 1,2,3,(10,3),4_2,5_2,7_1,8_2	8_2	8_2 -> 7_1 8_2 -> (10,3)
9^2	Z -> 8_2	3 ->(10，3)	9_2 -> 1,2,3,(10,3),4_2,5_2,7_1,8_2	9_2	9_2-> 8_2 9_2 -> (10,3)

4、动态切片

- 切片计算示例

执行历史： <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>

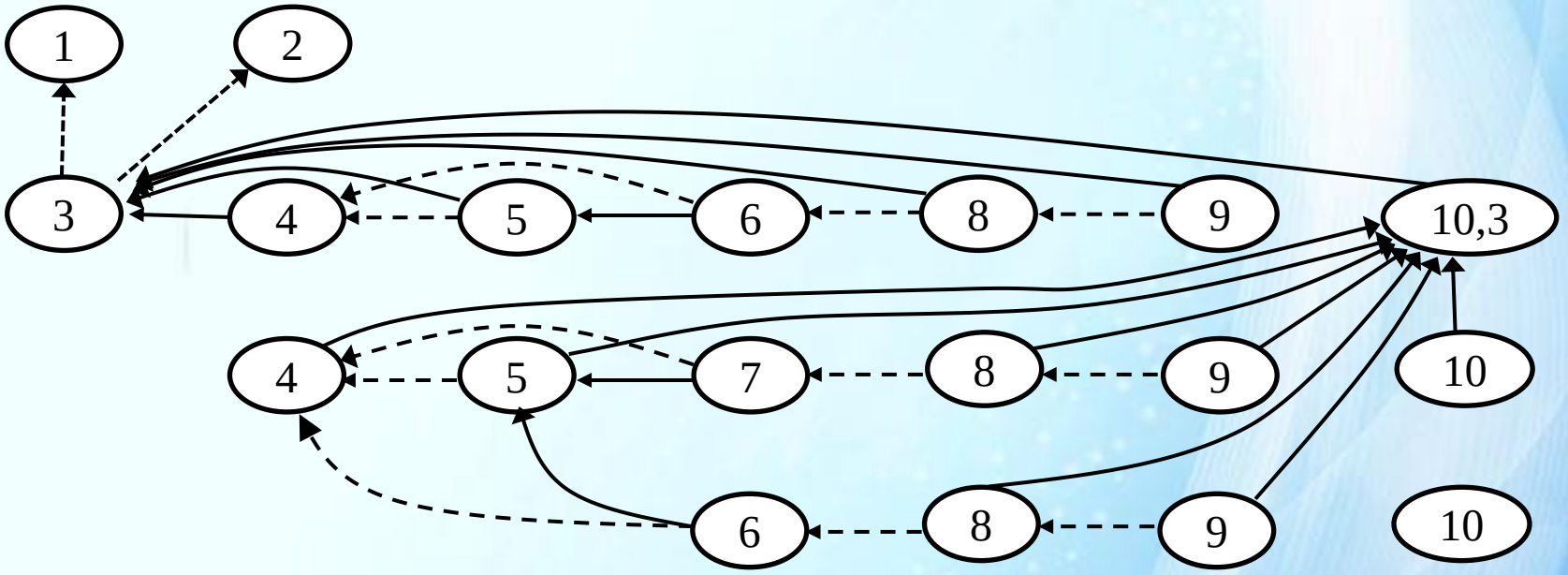
```
1:      N = getchar();
2:      I = 1;
3:      while(I <= N){
4:          x = getchar();
5:          if (x < 0)
6:              y = f1(x);
            else
7:              y = f2(x);
8:          z = f3(y);
9:          printf("%d\n",z);
10:         I = I + 1;
        }
```

序号	D	C	R	新节点	新建边
10^2	I -> (10,3)				
3^3 4^3 5^2					
6^2	X -> 4_2	5-> 5_2	6_2 -> 1,2,3,(10,3),4_2,5_2,6_2	6_2	6_2 -> 4_2 6_2 -> 5_2
8^3	Y-> 6_2	3-> (10, 3)	8_3 -> 1,2,3,(10,3),4_2,5_2,6_2,8_3	8_3	8_3 -> 6_2 8_3 -> (10,3)
9^3	Z -> 8_3	3-> (10, 3)	9_3 -> 1,2,3,(10,3),4_2,5_2,6_2,8_3,9_3	9_3	9_3 -> 8_3 9_3 -> (10,3)

4、动态切片

- 切片准则 < 9^3 , z > 输入 N = 3, x分别为-4，3和-2

执行历史： <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^2, 6^2, 8^3, 9^3, 10^3, 3^4>



4、动态切片

- 小结
  - 方法一：
    - 基于程序依赖图的静态切片算法进行扩展，删除不在执行历史中的节点
  - 方法二：
    - 考虑在动态运行中发生依赖的边
  - 方法三：
    - 考虑节点的不同运行实例，构造动态依赖图
  - 方法四：
    - 基于动态依赖图中节点等价性进行化简

5、程序切片应用

- 切片工具
  - 静态切片
    - 免费：Angr，Dyninst, Eclipse Indus，Frama-C
    - 商业软件：GammaTech CodeSurfer
  - 动态切片
    - 免费：
      - JSlicer，WET(An Advanced Infrastructure for Generation, Storage, and Analysis of Program Execution Traces)
      - Panda slicer

5、程序切片应用

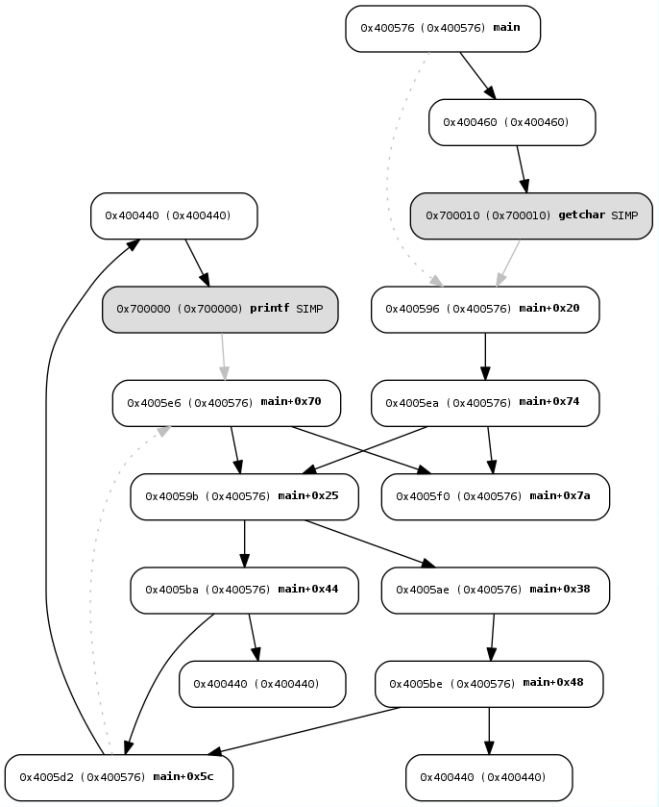
- 切片工具
  - Angr
    - 面向二进制代码的软件分析平台 （https://docs.angr.io/built-in-analyses/backward_slice）

```
>>> import angr
# Load the project
>>> b = angr.Project("examples/fauxware/fauxware", load_options={"auto_load_libs":

# Generate a CFG first. In order to generate data dependence graph afterwards, you
# - keep all input states by specifying keep_state=True.
# - store memory, register and temporary values accesses by adding the angr.optior
# Feel free to provide more parameters (for example, context_sensitivity_level) fo
# recovery based on your needs.
>>> cfg = b.analyses.CFGEmulated(keep_state=True,
...                              state_add_options=angr.sim_options.refs,
...                              context_sensitivity_level=2)

# Generate the control dependence graph
>>> cdg = b.analyses.CDG(cfg)

# Build the data dependence graph. It might take a while. Be patient!
>>> ddg = b.analyses.DDG(cfg)

# See where we wanna go... let's go to the exit() call, which is modeled as a
# SimProcedure.
>>> target_func = cfg.kb.functions.function(name="exit")
# We need the CFGNode instance
>>> target_node = cfg.get_any_node(target_func.addr)

# Let's get a BackwardSlice out of them!
# `targets` is a list of objects, where each one is either a CodeLocation
# object, or a tuple of CFGNode instance and a statement ID. Setting statement
# ID to -1 means the very beginning of that CFGNode. A SimProcedure does not
# have any statement, so you should always specify -1 for it.
>>> bs = b.analyses.BackwardSlice(cfg, cdg=cdg, ddg=ddg, targets=[ (target_node,

# Here is our awesome program slice!
>>> print(bs)
```

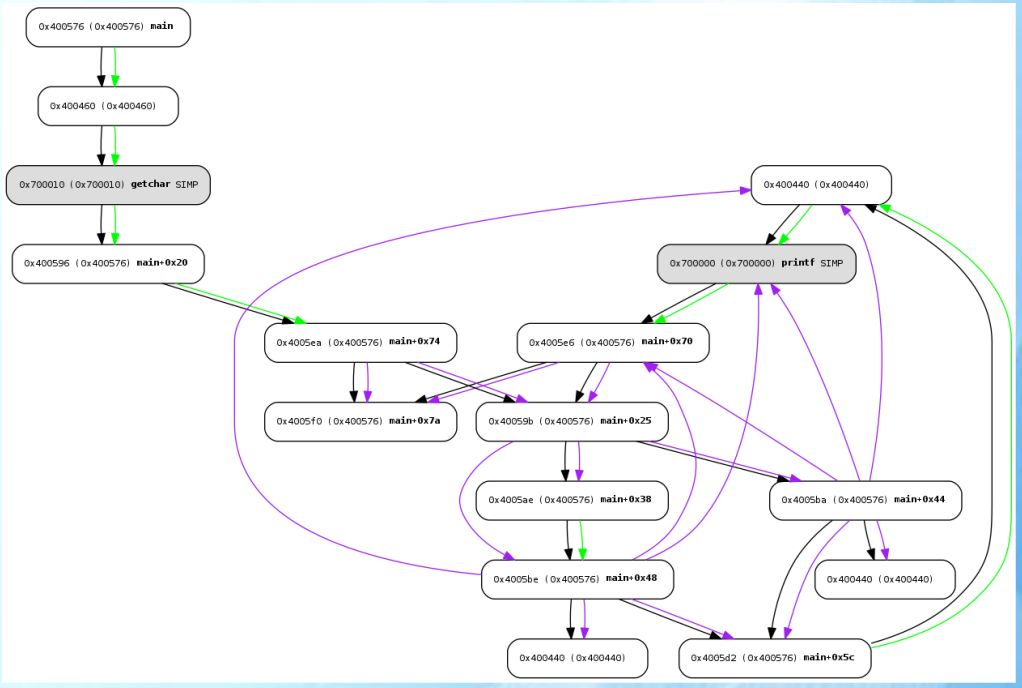
angr/analyses/backward_slice.py

5、程序切片应用

- 切片工具
  - Angr



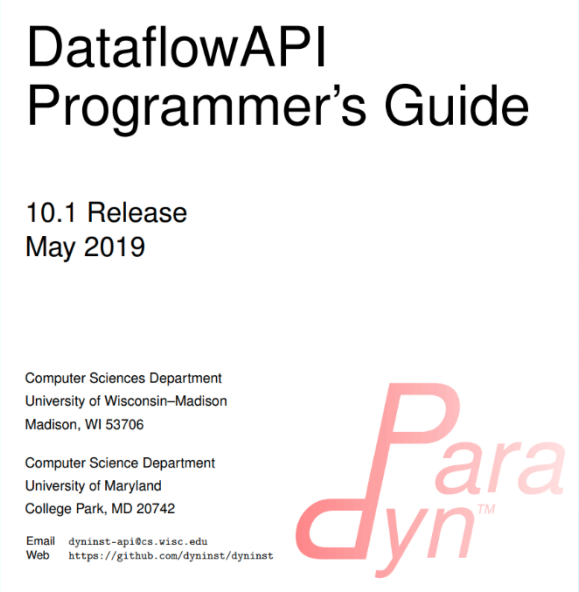
CDG



DDG

5、程序切片应用

- 切片工具
  - –Dyninst Dataflow
    - 面向二进制代码的软件分析平台（https://github.com/dyninst/dyninst）
    - 程序切片包含在其Dataflow模块中

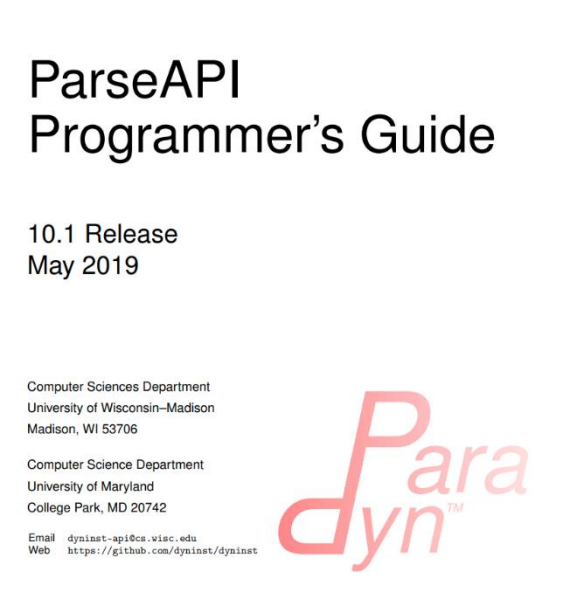

DataflowAPI
Programmer's Guide

10.1 Release
May 2019

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web https://github.com/dyninst/dyninst


ParseAPI
Programmer's Guide

10.1 Release
May 2019

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web https://github.com/dyninst/dyninst

5、程序切片应用

- 切片工具
  - Dyninst Dataflow

```
void SliceTarget(Function *f, Block *b, Address addr) {
    const unsigned char *buf = (const unsigned char*) b->obj()->cs()->getPtrToInstruction(addr);
    InstructionDecoder dec(buf,InstructionDecoder::maxInstructionLength,b->obj()->cs()->getArch());
    Instruction insn = dec.decode();
    AssignmentConverter ac(true,true);
    vector<Assignment::Ptr> assignments;
    ac.convert(insn, addr, f, b, assignments);

    Assignment::Ptr pcAssign;
    for (auto ait = assignments.begin(); ait != assignments.end(); ++ait) {
        const AbsRegion &out = (*ait)->out();
        if (out.absloc().type() == Absloc::Register) {
            pcAssign = *ait;
            break;
        }
    }
    Slicer s(pcAssign, b, f);
    ControlPred mp;
    GraphPtr slice = s.backwardSlice(mp);
    fprintf(stderr,"Graph Size: %u\n",slice->size());
    NodeIterator node_it, node_begin_it, node_end_it;
    slice->allNodes(node_begin_it,node_end_it);
    for(node_it = node_begin_it; node_it != node_end_it; node_it++){
        fprintf(stderr,"\t node: %p\n",(*node_it)->addr());
    }
}
```

5、程序切片应用

- Dyninst Dataflow

```
int main() {
    int x, y, z;
    int i=0;
    z = 0;
    y = getchar();
    for(;i<100;i++){
        if (i%2 == 1)
            x += y*i;
        else
            z += 1;
        printf("%d\n",x);
        printf("%d\n",z);
    }
}
```

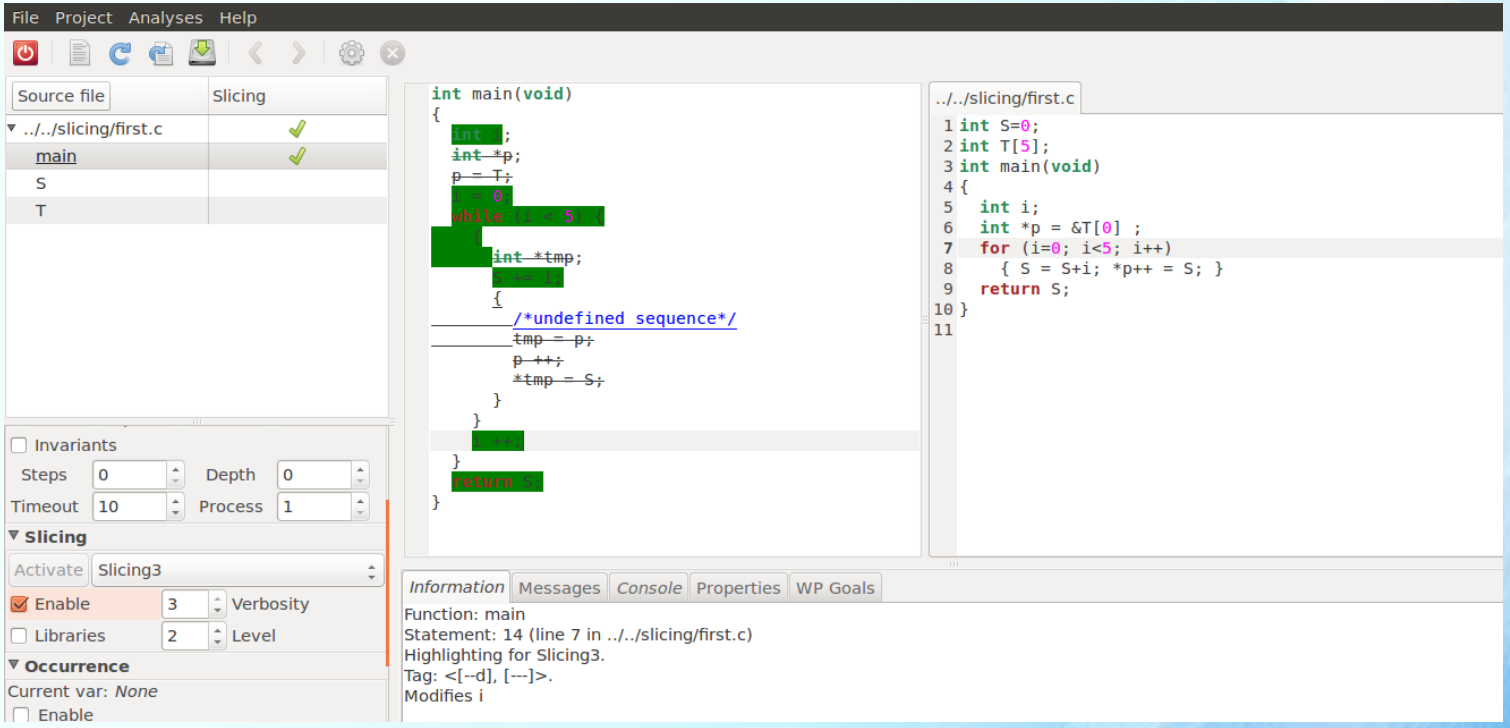
```
sample path ../test_elf/test
4005c1
Graph Size: 10
        node: 0x400576
        node: 0x400577
        node: 0x400596
        node: 0x4005e6
        node: 0x4005ae
        node: 0x4005c1
        node: 0x40057e
        node: 0x4005be
        node: 0x4005b5
        node: 0x4005b1
Analysis Done!
```

```
0000000000400576 <main>:
  400576:       55                      push   %rbp
  400577:       48 89 e5                mov    %rsp,%rbp
  40057a:       48 83 ec 10             sub    $0x10,%rsp
  40057e:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400585:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
  40058c:       b8 00 00 00 00          mov    $0x0,%eax
  400591:       e8 ca fe ff ff          callq  400460 <getchar@plt>
  400596:       89 45 fc                mov    %eax,-0x4(%rbp)
  400599:       eb 4f                   jmp    4005ea <main+0x74>
  40059b:       8b 45 f8                mov    -0x8(%rbp),%eax
  40059e:       99                      cltd
  40059f:       c1 ea 1f                shr    $0x1f,%edx
  4005a2:       01 d0                   add    %edx,%eax
  4005a4:       83 e0 01                and    $0x1,%eax
  4005a7:       29 d0                   sub    %edx,%eax
  4005a9:       83 f8 01                cmp    $0x1,%eax
  4005ac:       75 0c                   jne    4005ba <main+0x44>
  4005ae:       8b 45 fc                mov    -0x4(%rbp),%eax
  4005b1:       0f af 45 f8             imul   -0x8(%rbp),%eax
  4005b5:       01 45 f0                add    %eax,-0x10(%rbp)
  4005b8:       eb 04                   jmp    4005be <main+0x48>
  4005ba:       83 45 f4 01             addl   $0x1,-0xc(%rbp)
  4005be:       8b 45 f0                mov    -0x10(%rbp),%eax
  4005c1:       89 c6                   mov    %eax,%esi
  4005c3:       bf 84 06 40 00          mov    $0x400684,%edi
  4005c8:       b8 00 00 00 00          mov    $0x0,%eax
  4005cd:       e8 6e fe ff ff          callq  400440 <printf@plt>
```

```
  4005e1:       e8 5a fe ff ff          callq  400440 <printf@plt>
  4005e6:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
  4005ea:       83 7d f8 63             cmpl   $0x63,-0x8(%rbp)
```

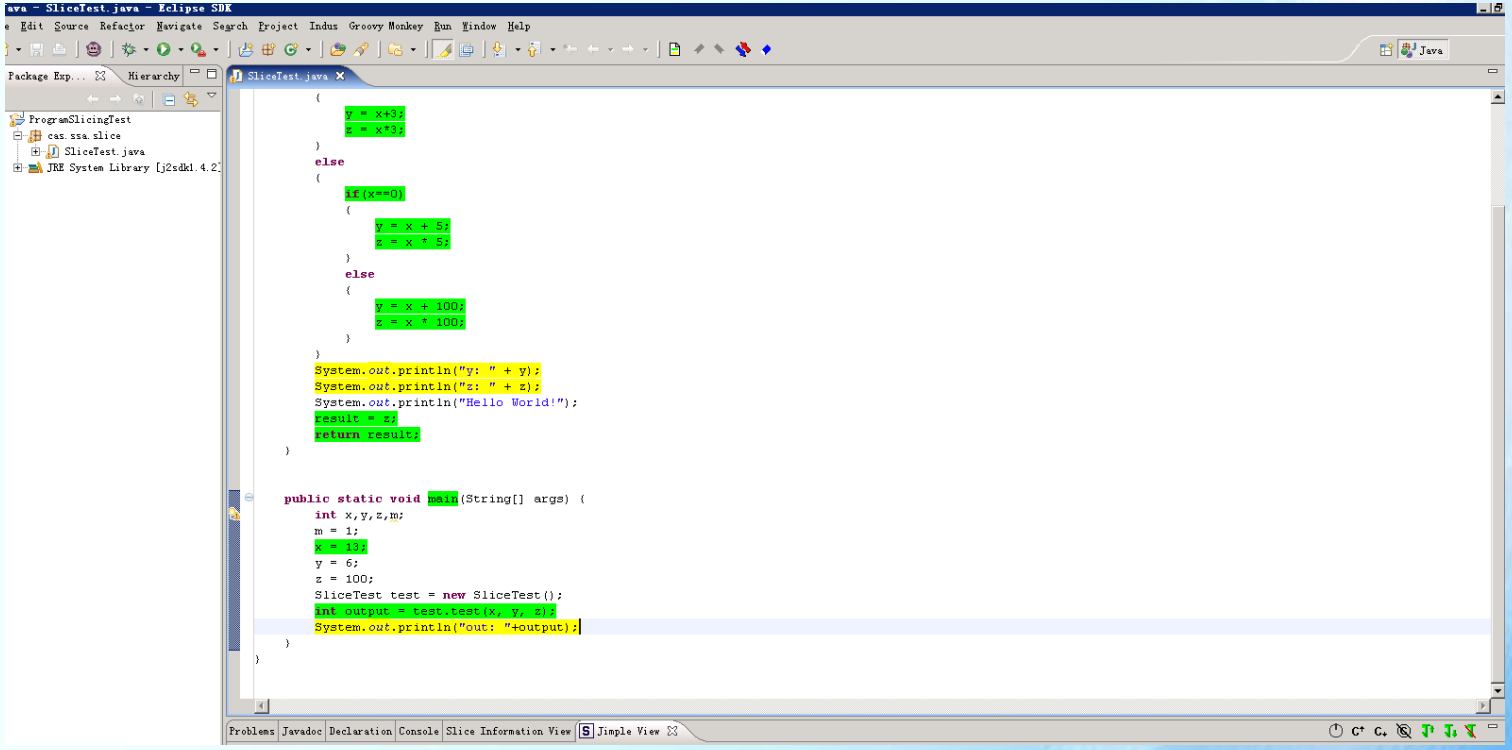
5、程序切片应用

- 切片工具
  - –Frama-C
    - 面向C源代码的开源可扩展分析平台 （http://frama-c.com/）
    - 程序切片是其插件之一



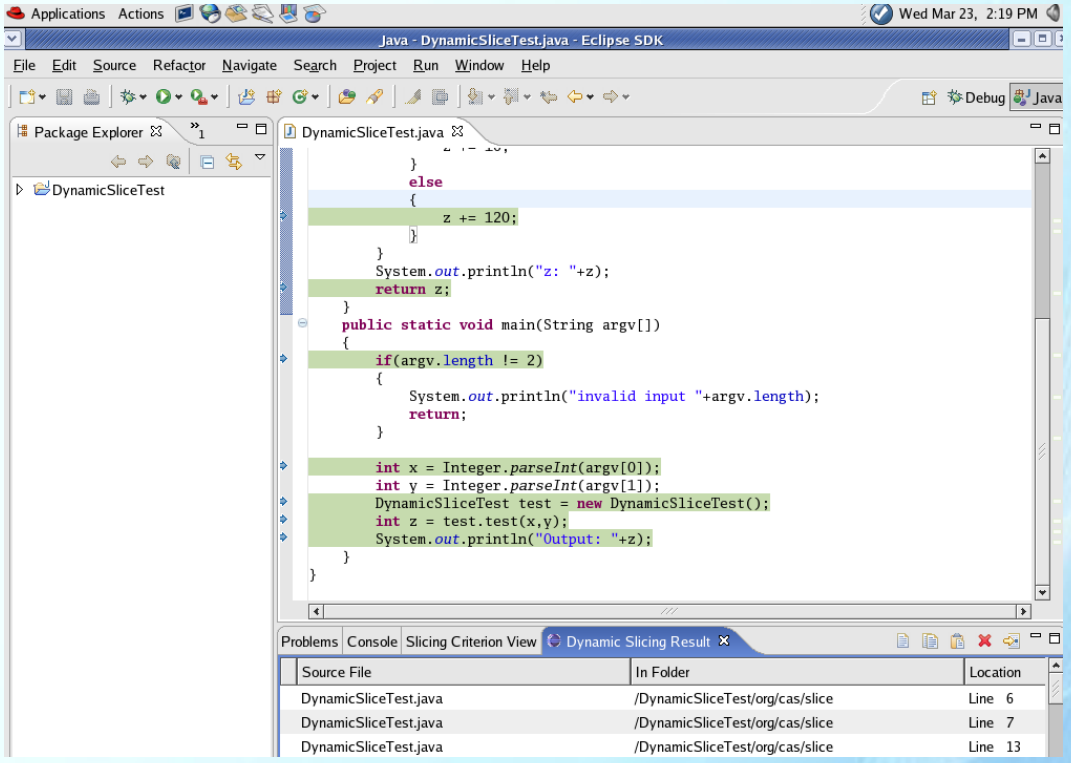
5、程序切片应用

- 切片工具
  - –Indus
    - 面向Java程序的静态切片工具，可作为Eclipse插件使用
    - http://indus.projects.cis.ksu.edu/



5、程序切片应用

- 切片工具
  - –JSlicer (http://**jslice**.sourceforge.net/)
    - 面向Java程序的动态切片工具，可作为Eclipse插件使用
    - 通过在Java虚拟机中嵌入分析代码来跟踪动态执行过程



5、程序切片应用

- 参考书目
  - 学术论文
    - Mark Weiser. 1981. Program slicing. In Proceedings of the 5th international conference on Software engineering (ICSE '81). IEEE Press, 439–449
    - Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. 1993. Debugging with dynamic slicing and backtracking. Softw. Pract. Exper. 23, 6 (June 1993), 589–616.
    - Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. PLDI '90.
  - 《程序切片技术及其应用》李必信等编著
  - 《静态程序分析》李樾、谭添等
    - https://github.com/RangerNJU/Static-Program-Analysis-Book

谢谢