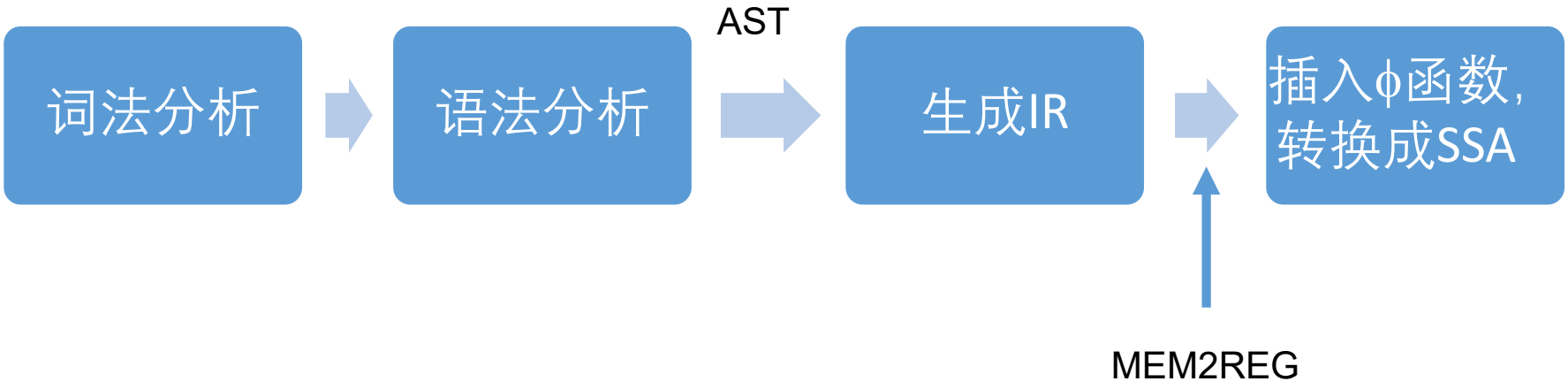
并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室

培训二：静态单一赋值*SSA*

翻译成IR



IR

- 局部变量生成为 alloca/load/store 形式
  - 用 alloca 指令来“声明”变量，得到一个指向该变量的指针
  - 用 store 指令来把值存在变量里
  - 用 load 指令来把值读出为 SSA value
- 临时变量
  - 通过加入下标，满足单一赋值

IR

```
1 void work(int a, int b, int n)
2 {
3     int i=0;
4     a = b+n+2;
5     for(i=0;i<n;i++)
6         a = a+i;
7     if(n<b)
8         a = b;
9     return;
10 }
```

```
define dso_local void @work(i32 noundef %0, i32 noundef %1, i32 noundef
%2) #0 {
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  store i32 %0, ptr %4, align 4
  store i32 %1, ptr %5, align 4
  store i32 %2, ptr %6, align 4
  store i32 0, ptr %7, align 4
  %8 = load i32, ptr %5, align 4
  %9 = load i32, ptr %6, align 4
  %10 = add nsw i32 %8, %9
  %11 = add nsw i32 %10, 2
  store i32 %11, ptr %4, align 4
  store i32 0, ptr %7, align 4
  br label %12

12:                                               ; preds = %20, %3
  %13 = load i32, ptr %7, align 4
  %14 = load i32, ptr %6, align 4
  %15 = icmp slt i32 %13, %14
   br i1 %15, label %16, label %23

16:                                               ; preds = %12
  %17 = load i32, ptr %4, align 4
  %18 = load i32, ptr %7, align 4
  %19 = add nsw i32 %17, %18
  store i32 %19, ptr %4, align 4
  br label %20

20:                                               ; preds = %16
%21 = load i32, ptr %7, align 4
%22 = add nsw i32 %21, 1
store i32 %22, ptr %7, align 4
br label %12, !llvm.loop !6
23:                                               ; preds = %12
%24 = load i32, ptr %6, align 4
%25 = load i32, ptr %5, align 4
%26 = icmp slt i32 %24, %25
br i1 %26, label %27, label %29

27:                                               ; preds = %23
%28 = load i32, ptr %5, align 4
store i32 %28, ptr %4, align 4
br label %29

29:                                               ; preds = %27, %23
ret void
}
```

内容

1. 基本块
2. 控制流图
3. 必经关系
4. 定值-使用
5. SSA
6. LLVM SSA

1 基本块

- 基本块(Basic Block)
  - 基本块是一段只能从它的开始处进入，结束处离开的顺序代码序列
  - 基本块只有最后一条语句是分支语句，并且只有第一条语句是分支的目标
  - 在基本块内部，除了第一条指令外，每条指令都只有一个前驱；除了最后一条指令外，每条指令只有一个后继
- 使用基本块代替指令，可以减少分析的时间和空间开销

1.1 识别基本块

1. 首先，扫描程序，使用下列规则识别基本块的首语句：

（规则1） 程序/函数的第一条语句

（规则2） 分支语句的目标 (对大多数中间表示代码，分支的目标是带有标号的语句)

（规则3）任何紧跟在分支或者return语句后面的语句

1.1 识别基本块

1. 识别首语句
2. 每个首语句对应的基本块包括了首语句+下一个首语句之间（不包括下一个首语句）的语句

1.1 识别基本块

```
i=0;
for(i=0;i<n;i++)
    a = a+i;
if (n<b)
    a = b;
…
```

B1
```
(1) i = 0;
(2) goto <L2>;
```

B2
```
(3) <L1>:  a = a + i;
(4) i = i + 1;
```

B3
```
(5) <L2>:  if (i < n) goto <L1>;
                      else goto <L3>;
```

B4
```
(6) <L3>:  if (n < b) goto <L4>;
                      else goto <L5>;
```

B5
```
(7) <L4>:  a = b;
```

B6
```
(8) <L5>:  ……
```

三地址代码

2 控制流图

- 控制流图（*control flow graph* ,CFG）是一个有向流图G=(*N*,*E*)，其中：
  - 结点*N*表示基本块
  - 边*E*表示程序的控制流向
- 首语句是第一条语句的称为开始结点（***start node***)
- CFG没有给出数据的任何信息。CFG中的边只表示程序可能走这条路径

2.1 构建控制流图

- 如果基本块B1和B2满足下面的条件之一，则存在一条从B1到B2的有向边
  - （1）从B1的最后一条语句可以跳转到B2的第一条语句
  - （2）B2紧跟在B1之后，并且B1的最后一条指令不是无条件转移

2.1 构建控制流图

B1
（1）i = 0;
（2）goto B3

B2
（3）<L1>: a = a + i;
（4）i = i + 1;

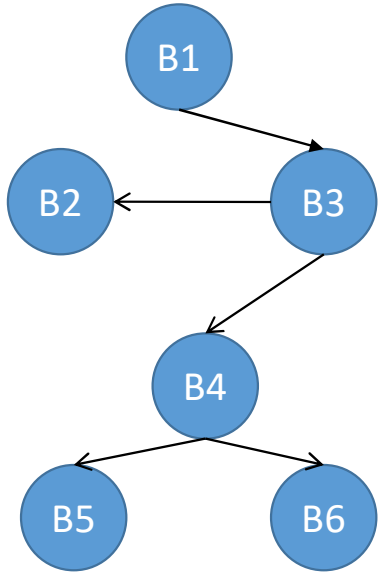
B3
（5）<L2>: if (i < n) goto B2;
else goto B4

B4
（6）<L3>: if (n < b) goto B5
else goto B6

B5
（7）<L4>: a = b;

B6
（8）<L5>: ……

条件1



2.1 构建控制流图

B1
```
（1）i = 0;
（2）goto B3
```

B2
```
（3）<L1>:  a = a + i;
（4）i = i + 1;
```

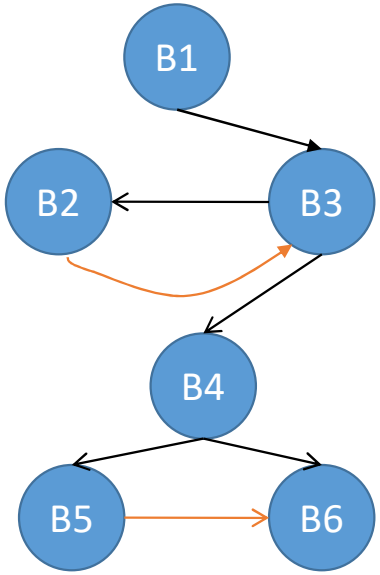
B3
```
（5）<L2>:  if (i < n) goto B2;
                  else goto B4
```

B4
```
（6）<L3>:  if (n < b) goto B5
                  else goto B6
```

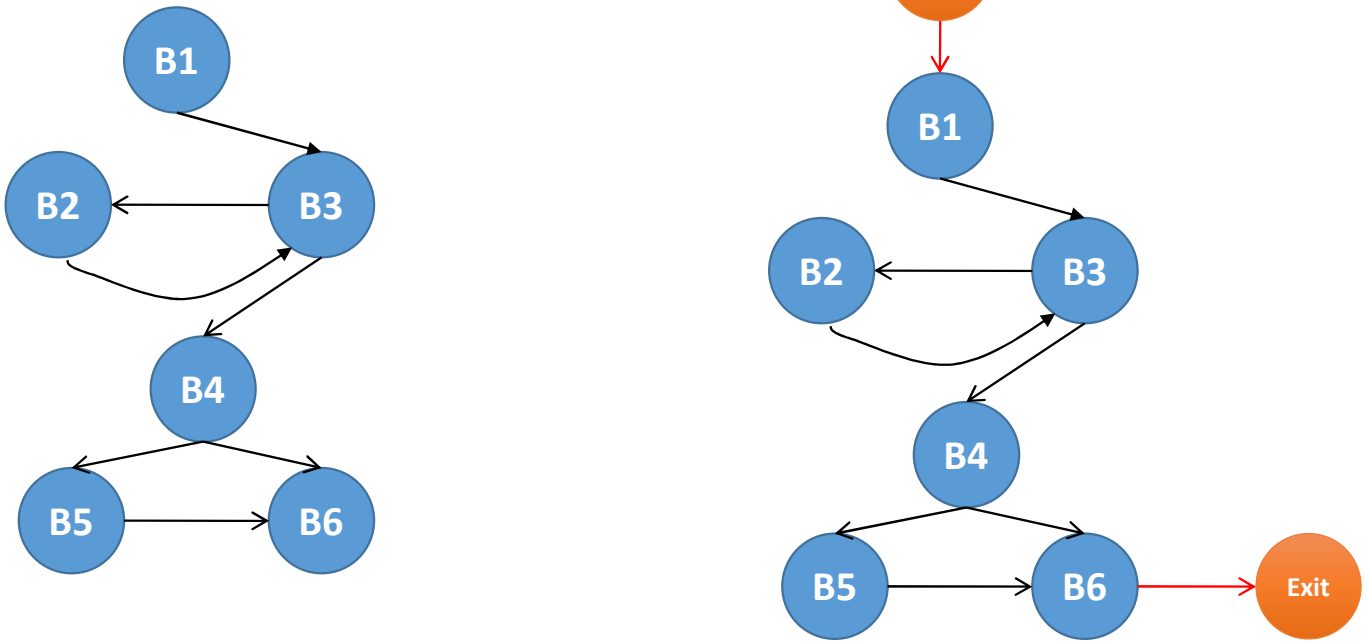
B5
```
（7）<L4>:  a = b;
```

B6
```
（8）<L5>:  ……
```

条件2



2.2 CFG中特殊结点



入口结点（Entry node）:一条entry到开始结点的边

出口结点（Exit Node）CFG图每一个从该函数出口的分支（没有后续的基本块）转向exit

1.3 前驱和后继

定义: 控制流图 $G = (N, E, s)$，其中：

N: 结点集合

E: 边的集合

s: 入口结点.

有 $a \in N, b \in N$.

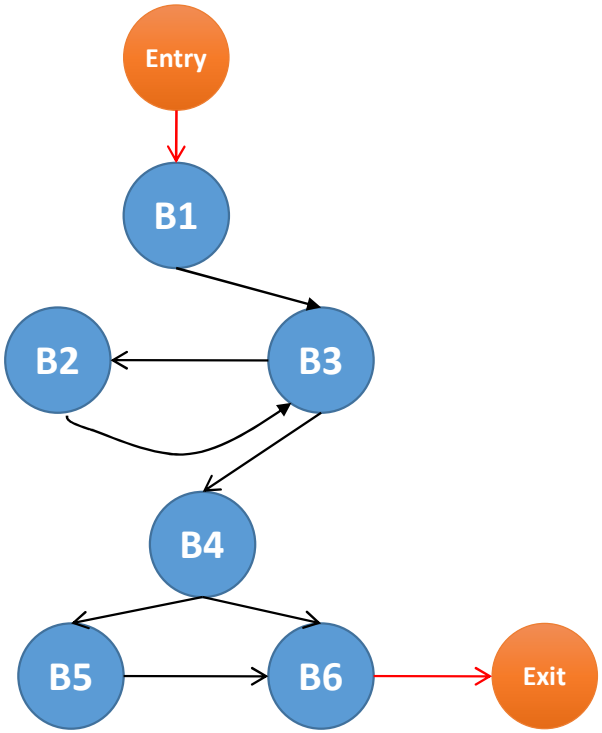
$$pred[b] = \{a \in N \mid \exists e \in E, e = a \rightarrow b\}$$

$$succ[b] = \{a \in N \mid \exists e \in E, e = b \rightarrow a\}$$

a 是分支结点 $if\ succ[a] > 1$

a 是汇合结点 $if\ pred[a] > 1$

2.3 前驱和后继



Pred(Entry) = ∅
Succ(Entry) = {B1}

Pred(B1) = {Entry}
Succ(Entry) = {B3}

Pred(B3) = {B1,B2}　汇合结点
Succ(B3) = {B2,B4}　分支结点

Pred(Exit) = {B6}
Succ(Exit) = ∅

```
i=0;
for(i=0;i<n;i++)
     a = a+i;
if (n<b)
     a = b;
…
```

讨论：
控制流图中，除了前驱后继，还可以获得什么样的程序执行信息？

3.1 必经结点

- 也称为支配结点

CFG中，如果从入口结点到结点b的每一条路径都经过结点a，则称a是b的必经结点。

b的所有必经结点构成b的必经结点集合dom(b)

必经结点关系是一种偏序关系：

1. 自反的, $a \in \text{dom}(a)$
2. 传递的, $a\ dom\ b$ & $b\ dom\ c$，则$a\ dom\ c$
3. 反对称的, if $a\ dom\ b$ & $b\ dom\ a$，则$a = b$

3.2 必经关系

1. 如果从入口结点s到b的每一条路径，都包括a，

   a 是b的必经结点，记作 $a \le b$

2. 如果$a \le b$ 且 $a \ne b$

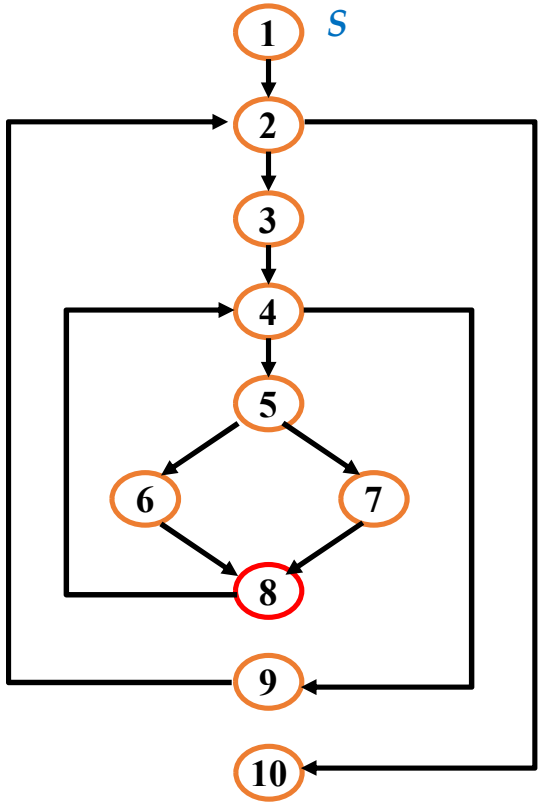
   a是b的严格必经结点， $a < b$

3. 如果$a<b$ 且不存在一个结点 c $\in$N，满足 a<c< b

   a是b的直接必经结点，$a <_i b$ ， 也记作a = idom (b)

4. 直接必经结点是唯一的

3.2 必经关系



必经关系:

{ (1, 1), (1, 2), (1, 3), (1,4) … ,(1,10)
(2, 2), (2, 3), …, (2, 10)
(3, 3),(3, 4),…, (3,9)
(4, 4),(4, 5),…, (4,9)
(5, 5),(5, 6) ,(5, 7),(5,8)
(6, 6), (7, 7) ,(8, 8) ,(9, 9) ,(10, 10)
}

直接必经关系:

$1 <_i 2, 2 <_i 3, 3 <_i 4, 4 <_i 5, 4 <_i 9,$
$5 <_i 6, 5 <_i 7, 5 <_i 8, 2 <_i 10$

必经结点集合:

dom (1) = {1}
dom (2) = {1, 2}
dom (3) = {1, 2, 3}
dom (10) = {1, 2, 10)

3.3 计算必经结点集合

- 基本思想
  - 结点b是b的必经结点。
  - 如果a是b的唯一前驱，则a是b的必经结点
  - 如果a是b所有前驱的必经结点，则a是b的必经结点

    $\forall p \in pred[b]$, if $a \le p$, 则 $a \le b$

3.3 计算必经结点集合

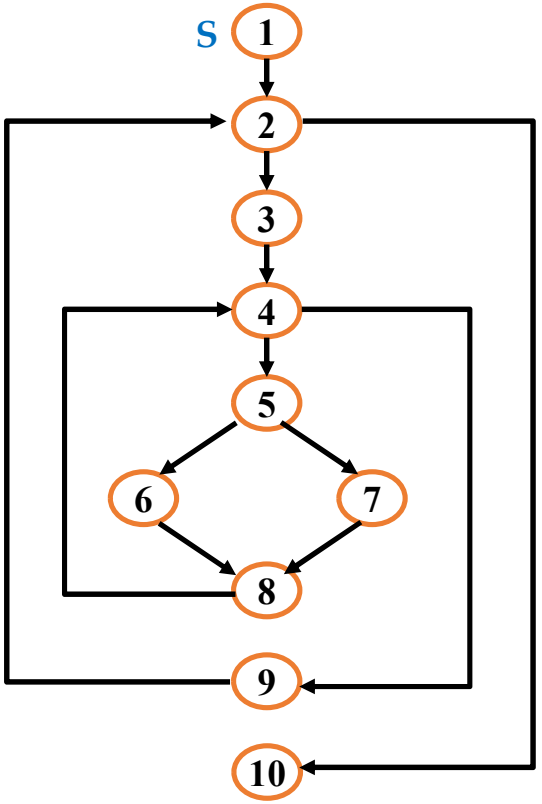
- 求结点a的必经结点集合dom[a]
  - 对于入口结点，dom[s] = {s}
  - 结点$a \neq s$，dom[a] = {a } ∪ (∩$p \in pred[a]$ dom[p]) for

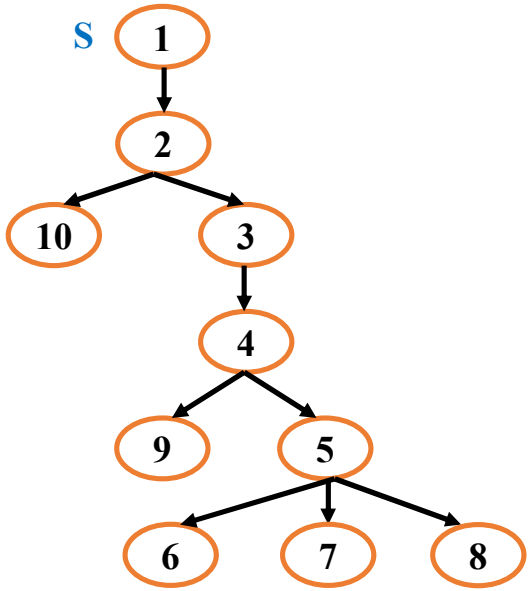
```
dom(s) = { s }
for n ∈ N –{ s } do dom(n) = N
repeat
   changed = false
   for n ∈ N –{s} {
     olddom = dom(n)
     dom(n) = {n} ∪ ∩p ∈PRED(n) dom(p)
      if dom(n) ≠ olddom then changed = true
   }
until changed = false
```

计算复杂性 : $O(N^2)$

3.4 必经结点树

- 必经结点树包含CFG所有结点，每个结点*a*的直接必经结点到*a*有一条有向边

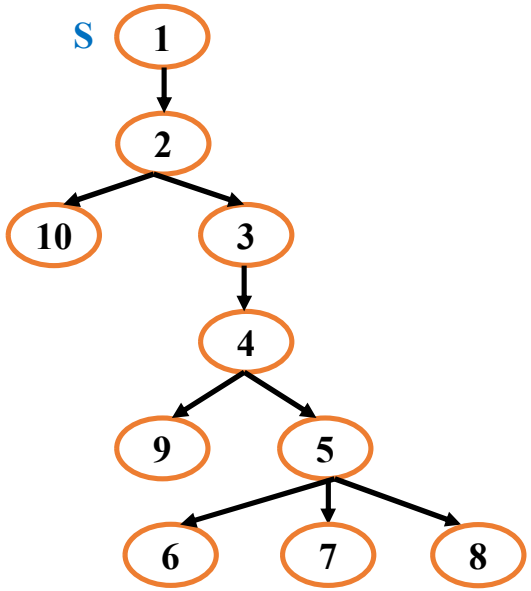




必经结点树

3.4 必经结点树

- 必经结点树包含CFG所有结点，每个结点*a*的直接必经结点到*a*有一条有向边
  - 入口结点s是根
  - 每个结点是其后代的必经结点



必经结点树

4 定值与使用

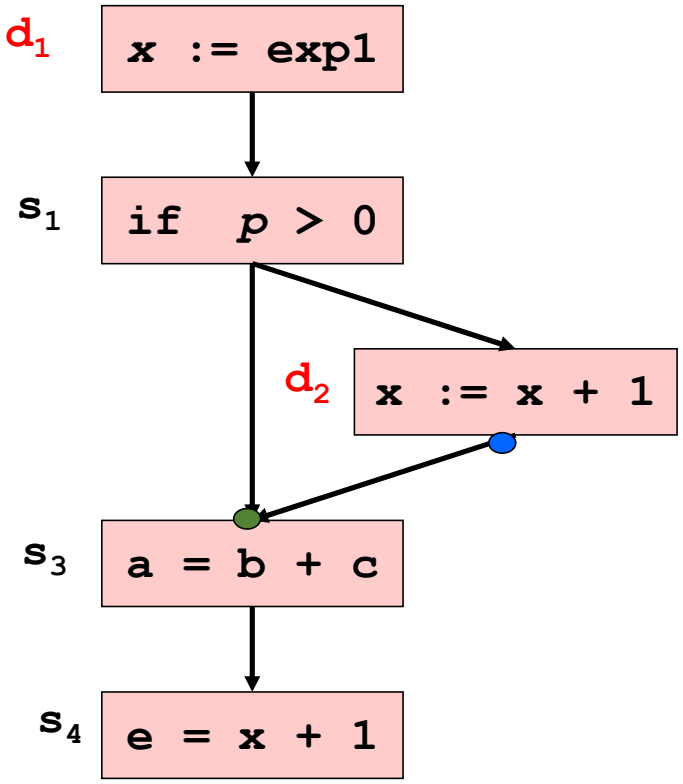
- 每个赋值就是一次定值

$S_k: V_1 = V_2 + V_3$

  - S_k 对变量V_1进行定值
  - S_k 使用 V_2 and V_3

4.1 到达定值

- 到达-定值（Reaching Definitions）
  - 变量*x*有一次定值d，如果存在一条从定值点开始的路径到达点p，并且沿着这条路径，定值d没有被其他定值杀死，那么就说定值d达到点p



d_1 到达 ●

d_1 不能到达 ●，被定值d_2杀死

4.2 到达-定值信息存储

- 位串表示
  - 一位表示一次定值
  - 位串长度=程序中定值次数
  - 集合的操作可以高效使用“位与”或者“位或”操作实现

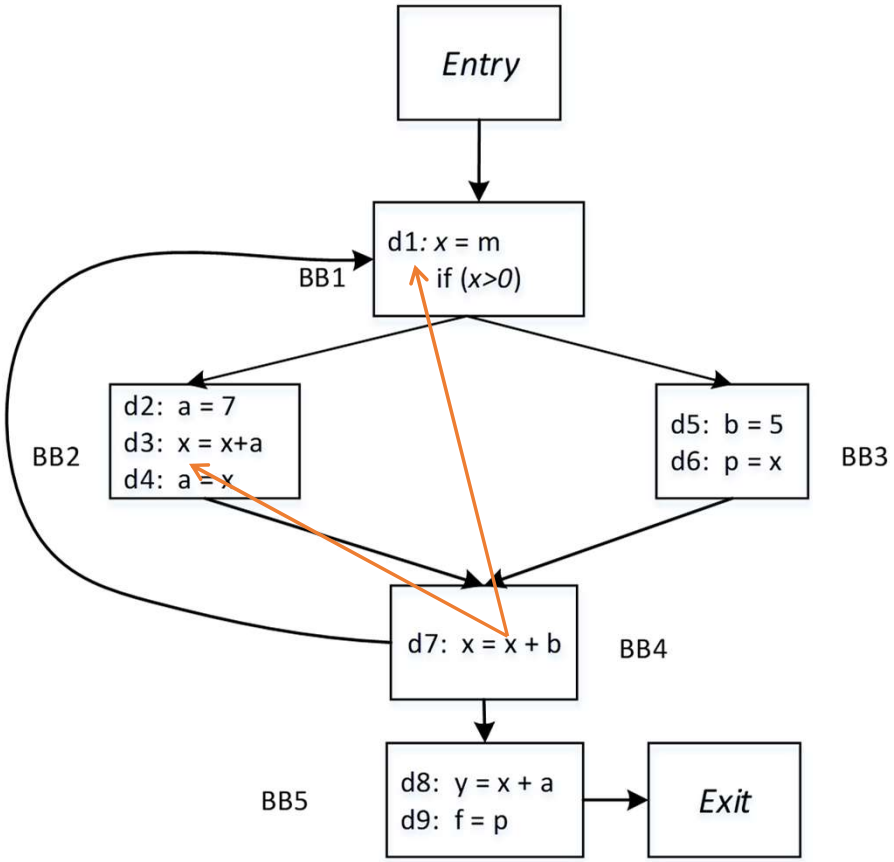
4.2 到达-定值信息存储

- 使用-定值链(Use-Definition Chain,UD链)
  - 一种稀疏的表示方式
  - 对变量的每一次使用，到达该使用的所有定值都保存在链表中

  如果使用在基本块B中
  - 如果B中在使用之前没有对v的定值，那么该次使用的UD链表包括in[B]中对v的所有定值
  - 否则，链表只包含一个元素，即在该次使用之前，对v的最后一次定值

4.2 到达-定值信息存储

- 使用-定值链(Use-Definition Chain，UD链)



- 基本块BB4中语句d7: x = x +b，到达该x的使用的定值为d1和d3
- BB2中语句d3: x=x+a，到达a的使用的定值只有d2

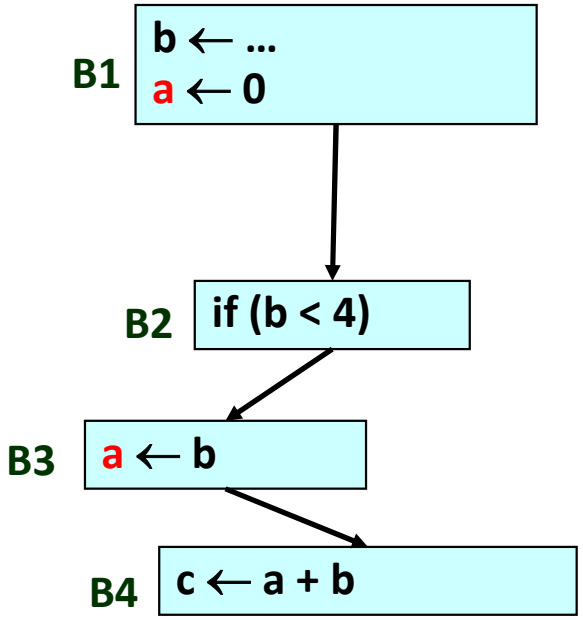
4.3 到达-定值分析讨论

- 位串表示 vs UD链
  - ？ 各自的优缺点
  - ？ 有没有更好的表示方法?
  - ？ 基于到达-定值分析，能够什么优化

5. Static Single Assignment

- A special form of IR code where each variable has only one definition in the program text
  - Each use of a variable is reached by exactly one assignment to that variable
  - A variable can be dynamically defined many times

5.1SSA Example



Is this code in SSA form?

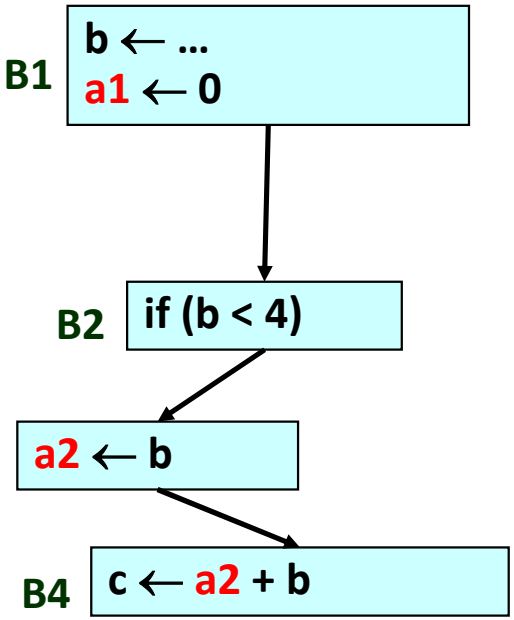
No, two definitions of ***a*** appear in the code (in B1 and B3).

How can we transform this code into a code in SSA form?

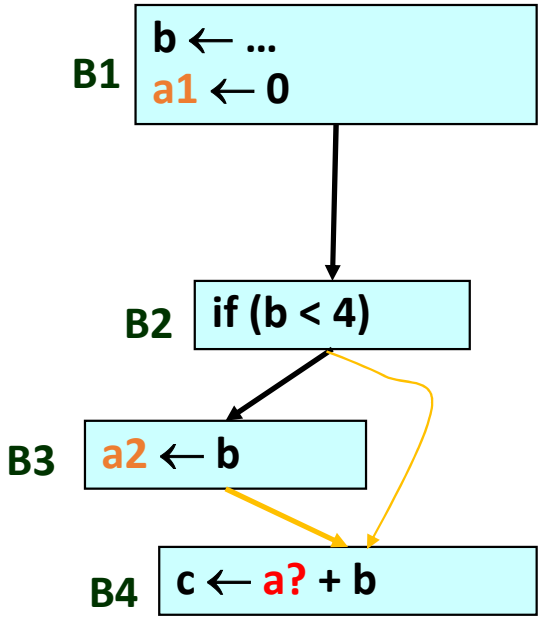
We can create two versions of ***a***, one for B1 and another for B3.

5.1 SSA Example

For straight-line code (e.g., basic block), it is easy to convert to SSA by renaming, e.g., adding subscripts.



5.1 SSA Example

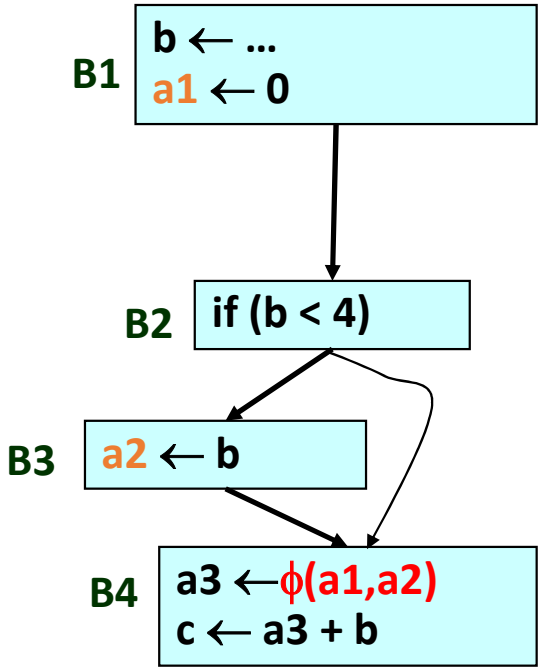


But which version should we use in B4 now?

We define a ϕ-function that "knows" which control path was taken to reach the basic block B4:

$$\varphi(a1,a2) = \begin{cases} a1 \text{ if arriving at B4 from B2} \\ a2 \text{ if arriving at B4 from B3} \end{cases}$$

5.1 SSA Example

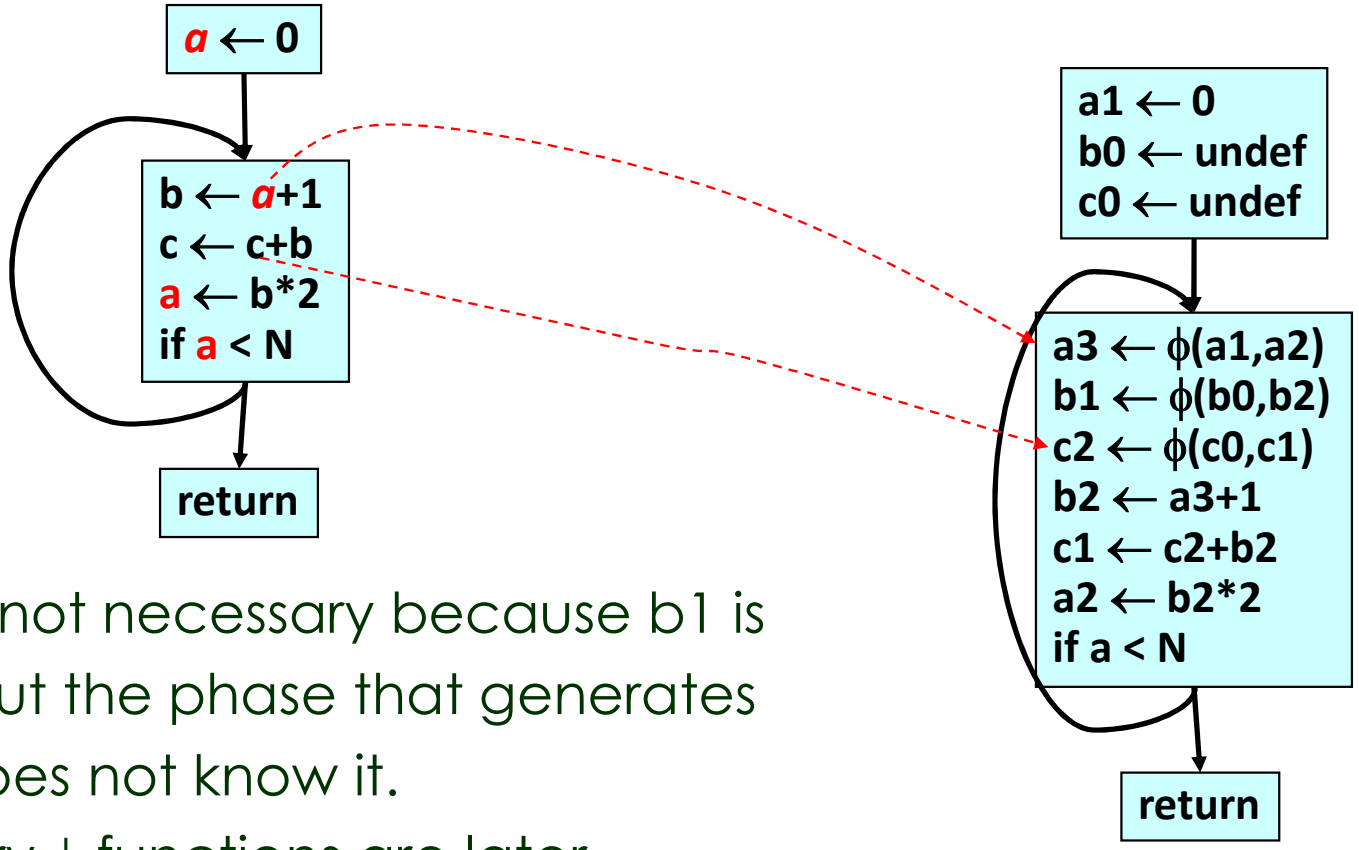


But which version should we use in B4 now?

We define a ϕ-function that "knows" which control path was taken to reach the basic block B4:

$$\varphi(a1,a2) = \begin{cases} a1 \text{ if arriving at B4 from B2} \\ a2 \text{ if arriving at B4 from B3} \end{cases}$$

5.1 SSA Example



- ϕ(b0,b2) is not necessary because b1 is never used. But the phase that generates ϕ-functions does not know it.
- Unnecessary ϕ functions are later eliminated by dead code elimination.

5.2 A Note on ϕ-function

- How can we implement a ϕ-function that "knows" which control path was taken?
  - The ϕ-function is used only to connect use to definitions during optimization, but is never implemented
  - If we must execute the ϕ-function, we can implement it by inserting *copy/move* instructions in all the control paths