

# Оптимизации на базе SSA

Оптимизации: LICM, SparseCond CP,  
DCE

Синявин А. В.

# Loop-invariant code motion

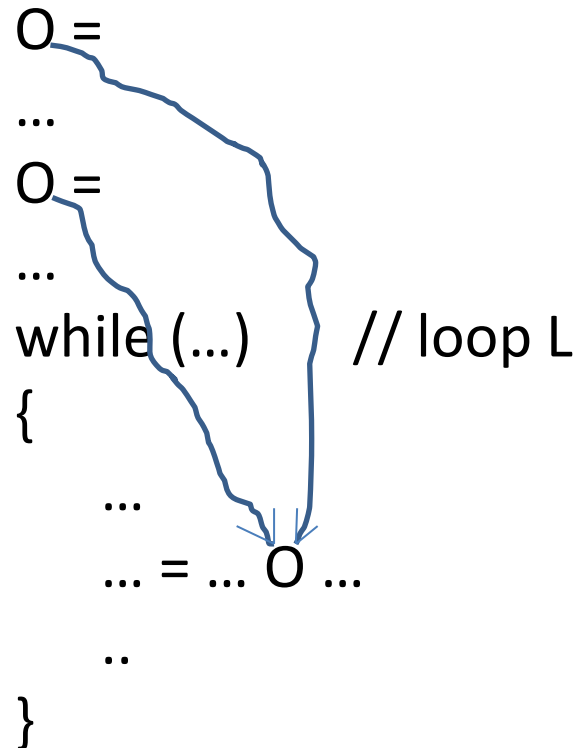
# Loop-invariant code motion

Инструкция  $p$  инвариантна относительно цикла  $L$ , если каждый операнд  $o$ :

- 1) является константным ( ***bool***  $IsConst(o)$  ) **или**
- 2) все определения, которые достигают данного использования, располагаются вне цикла  $L$  ( ***bool***  $IsOutDef(o, L)$  ) **или**
- 3)  $\exists!$  определение, которое достигает данного использования, расположено в цикле  $L$  и является инвариантным относительно  $L$  ( ***bool***  $IsInDef(o, L)$  )

# Loop-invariant code motion

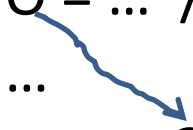
2) все определения, которые достигают данного использования, располагаются вне цикла L ( ***bool*** *IsOutDef*(*o*, *L*) )



# Loop-invariant code motion

3)  $\exists!$  определение, которое достигает данного использования, расположено в цикле  $L$  и является инвариантным относительно  $L$  ( ***bool*** *IsInDef*( $o, L$ ) )

```
while (...) // loop L
{
    ...
    O = ... // инвариантная инструкция
    ...
    ... = ... O ...
    ..
}
```



# Loop-invariant code motion

Алгоритм: Обнаружение инвариантных инструкций для цикла L

Вход: CFG g, только присваивания, цикл L

Выход: упорядоченный список инструкций InvarOrder

Метод:

```
map<stmt, bool> InstInvar;  
list<vertex> order = Breadth_Order();  
list<stmt> InvarOrder;  
bool change;  
foreach( vertex v in AllVert(g))  
    foreach( stmt s in v.stmts)  
        InstInvar[s] = false; /* помечаем все инструкции как неинвариантные */  
do  
{  
    change = false;  
    foreach( vertex v in order )  
        change = change || Mark_Block(v);  
}  
while(change);  
return InvarOrder;  
/* продолжение на следующем слайде */
```

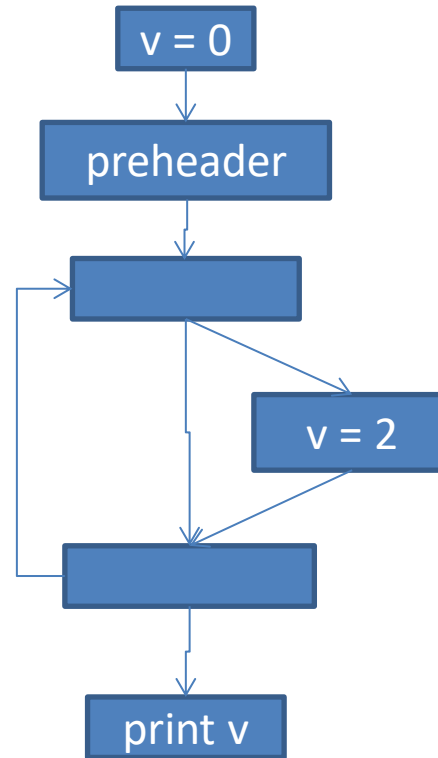
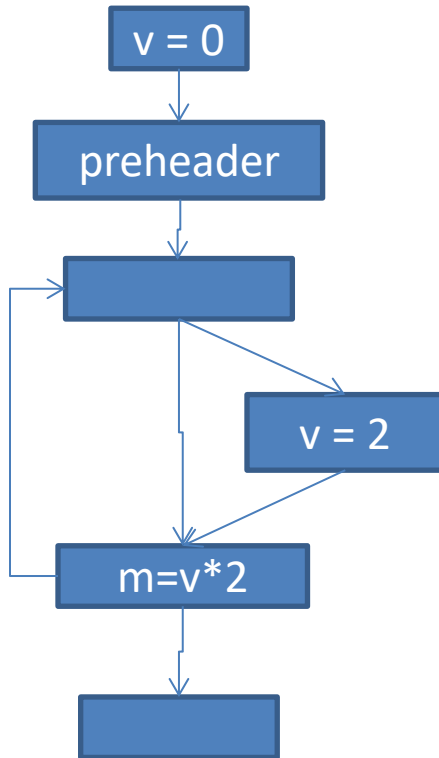
# Loop-invariant code motion

```
bool Mark_Block( vertex v )
{
    bool change = false;
    foreach( stmt s in v.stmts )
    {
        if ( !InstInvar[s] )
        {
            bool is_inv = true;
            foreach( operand o in s.rhs.opds )
            {
                if ( IsConst( o ) || IsOutDef( o, L ) || IsInDef(o, L) )
                    is_inv = is_inv && true;
                else
                    is_inv = is_inv && false;
            }
            InstInvar[s] = is_inv;

            if ( InstInvar[s] )
            {
                change = true;
                InvarOrder.push(s); /* добавление в конец списка */
            }
        }
    }
    return change;
}
```

# Loop-invariant code motion

Существуют две плохие ситуации:

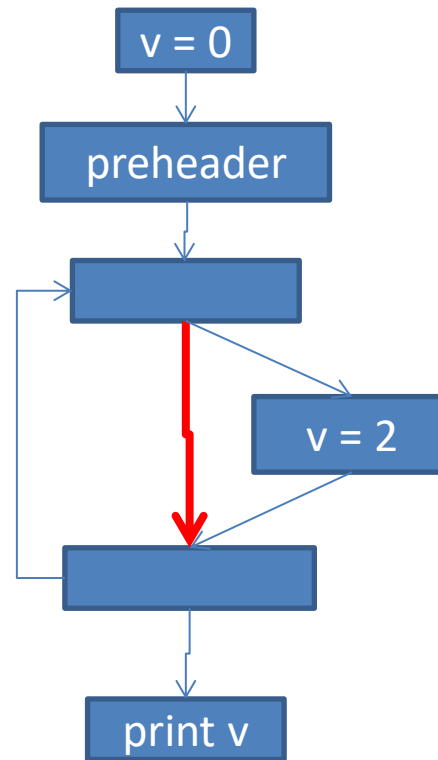
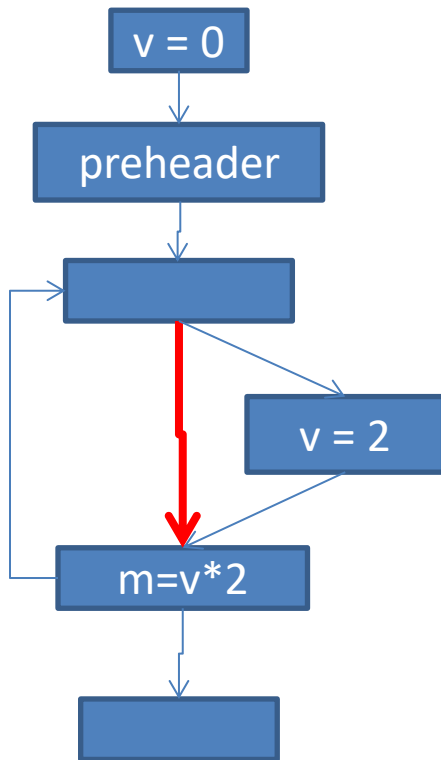


Инструкция “`v = 2`” является инвариантом, но если его переместить его preheader, то будет ошибка.



# Loop-invariant code motion

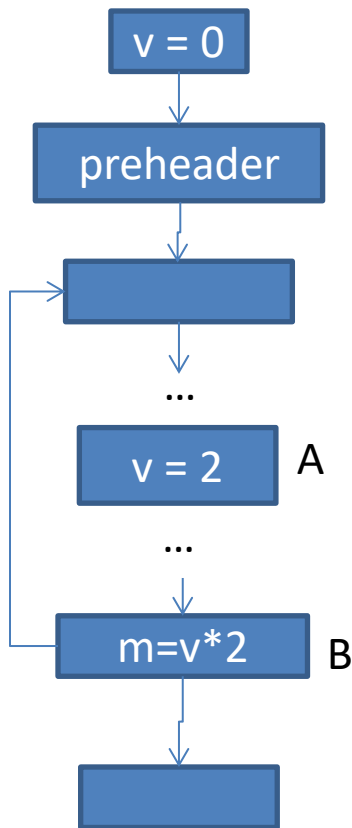
Существуют две плохие ситуации:



Инструкция “v = 2” является инвариантом, но если его переместить его preheader, то будет ошибка.

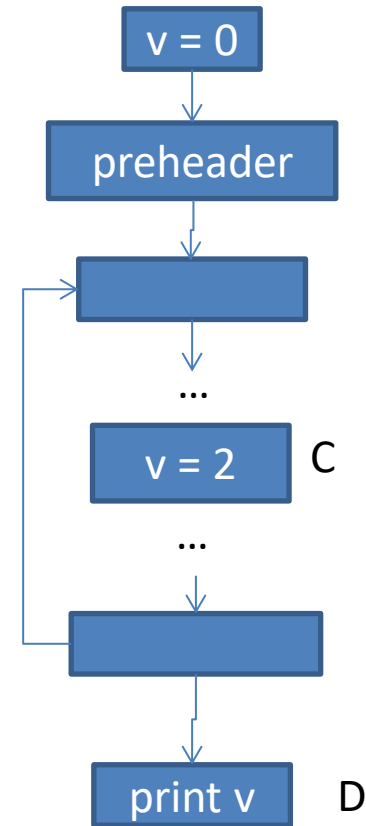
# Loop-invariant code motion

Как это “починить” ?



Потребуем:

A dom B



C dom D

# Loop-invariant code motion

Потребуем, чтобы:

- инструкция  $n$ , которая определяет некоторую переменную  $v$ , доминировала над всеми использованиями этой переменной внутри цикла ( ***bool** Dom\_Uses( $n$ )* )
- инструкция, которая определяет некоторую переменную, доминировала над всеми выходными блоками этого цикла ( ***bool** Dom\_Exits( $n$ )* )

# Loop-invariant code motion

Алгоритм: перенос инвариантных инструкций

Вход: CFG  $g$ , только присваивания, цикл  $L$ , список инструкций  $\text{InvarOrder}$

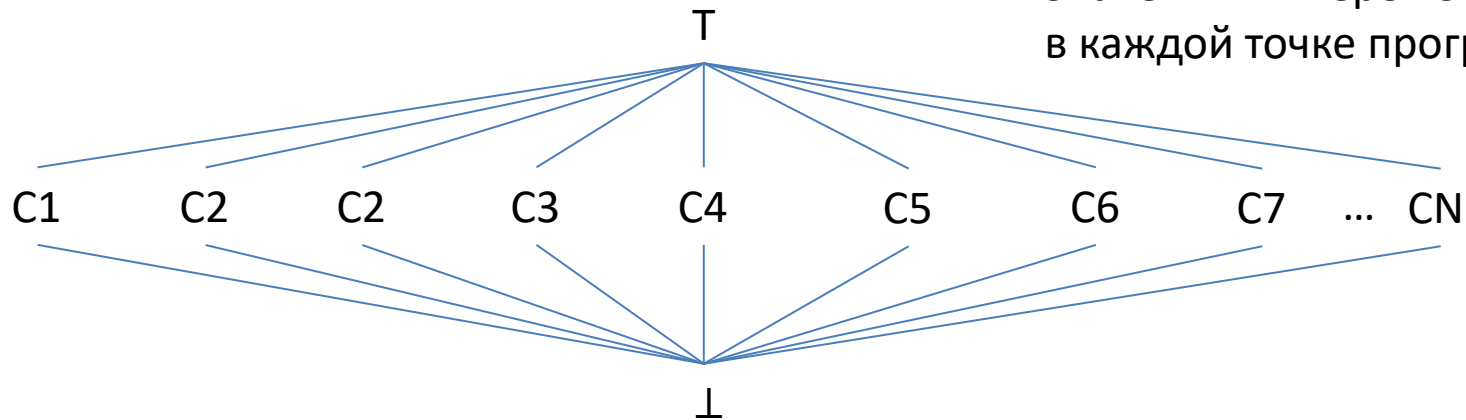
Выход: новый CFG  $g$

```
insert_preheader(g, L); /* добавляем preheader */
foreach(stmt s in InvarOrder)
{
    /* если истины условия, которые описаны выше */
    if ( Dom_Uses(s) && Dom_Exits(s) )
    {
        /* добавляем инструкцию в preheader */
        append_preheader(s);
        /* удаляем инструкцию из старого места */
        delete_inst(s);
    }
}
```

# Sparse conditional constant propagation

# Constant propagation

## Полурешётка SL



Ассоциируются со значениями переменных в каждой точке программы

$T$  неизвестное константное значение

$\perp$  переменное  $x$  – переменная

$C1 \dots CN$  константы  $Sl[x, p]$  – элемент полурешётки для переменной  $x$  в конкретной точке  $p$

# Constant propagation

Оператор “meet”

$$x \wedge T = x, \quad \forall x \in SL$$

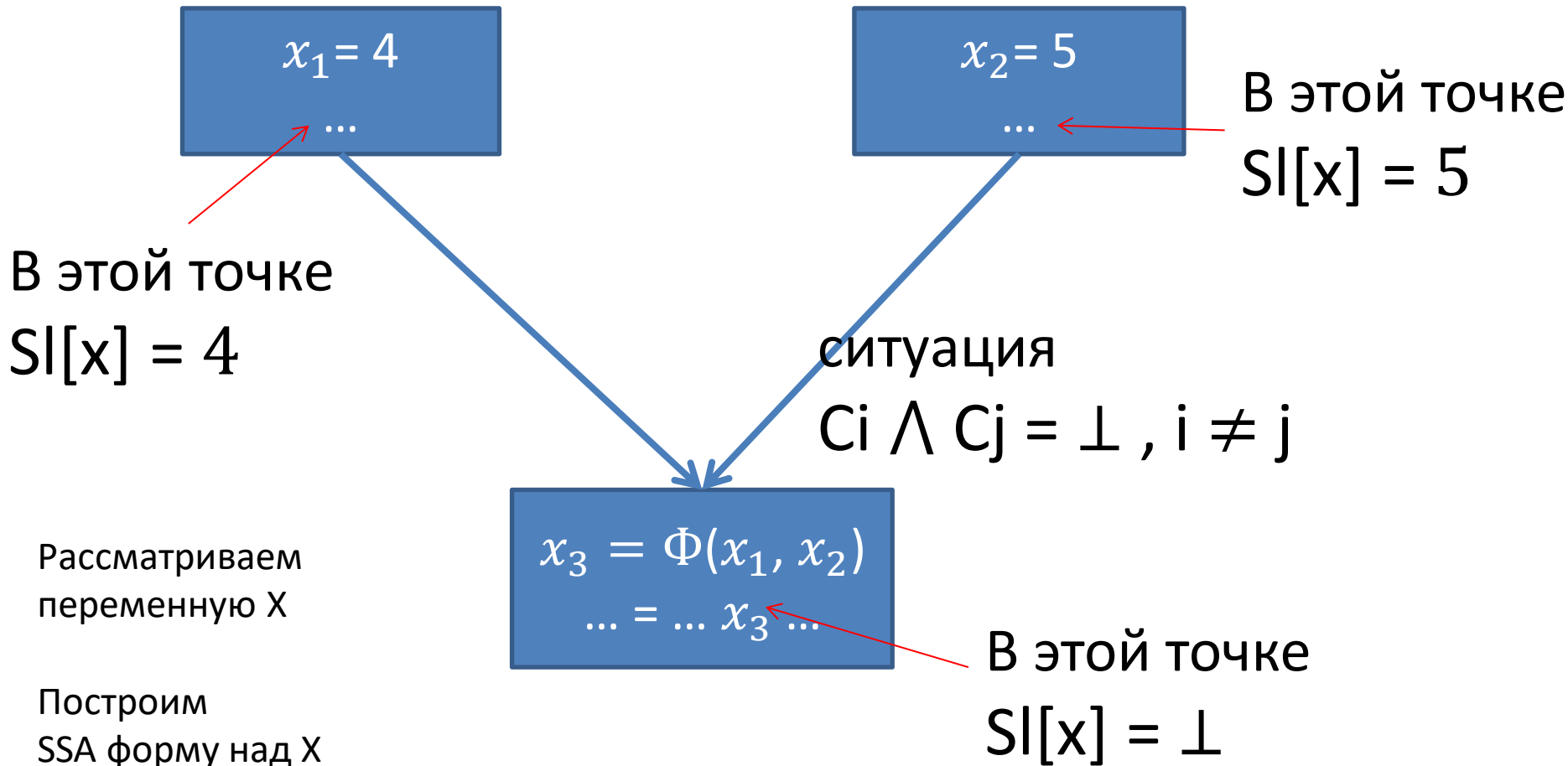
$$x \wedge \perp = \perp, \quad \forall x \in SL$$

$$C_i \wedge C_j = \perp, \quad i \neq j$$

$$C_i \wedge C_j = C_i, \quad i = j$$

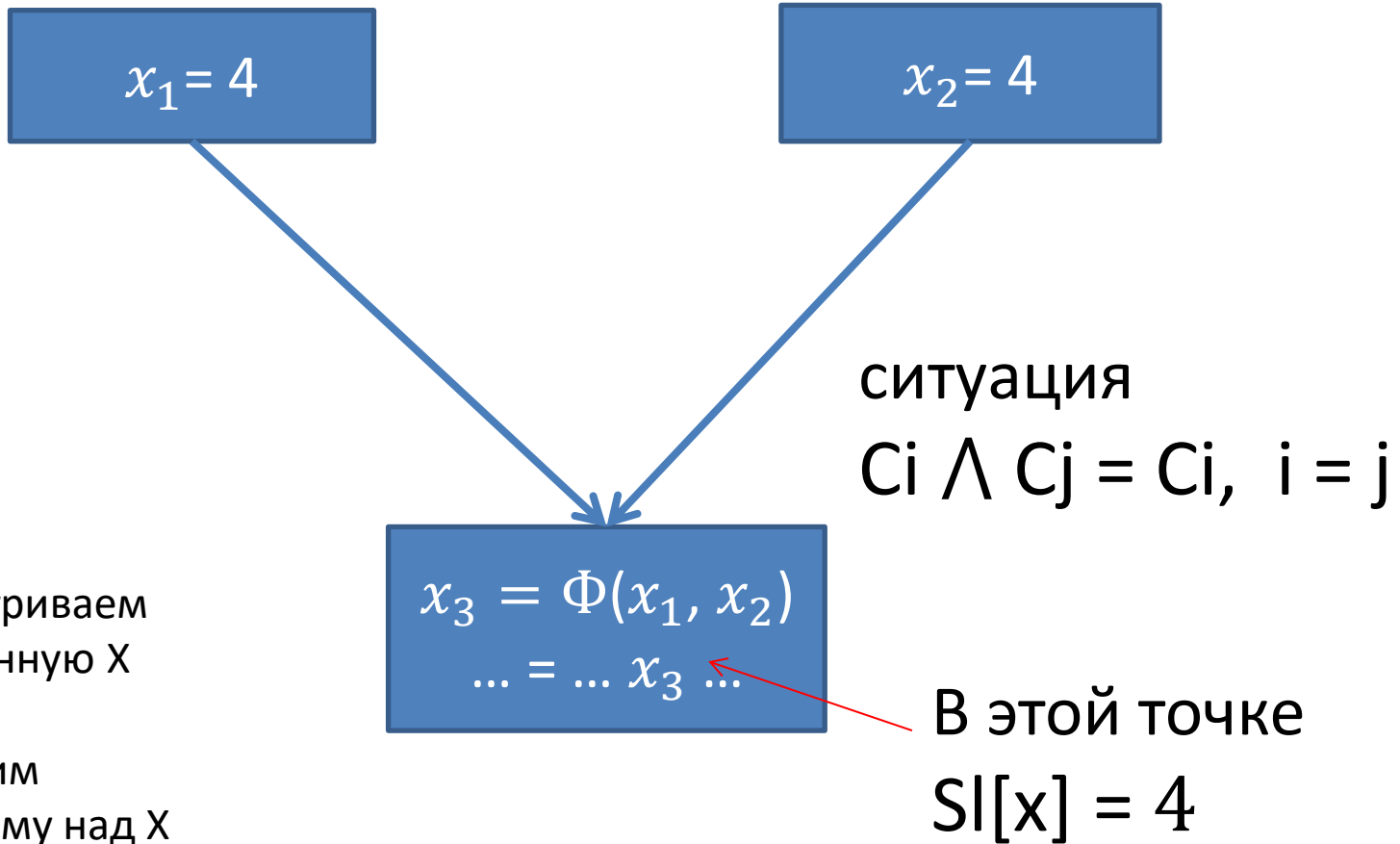
Будем использовать при обработке  $\phi$ -функций  
VisitPhi

# Constant propagation

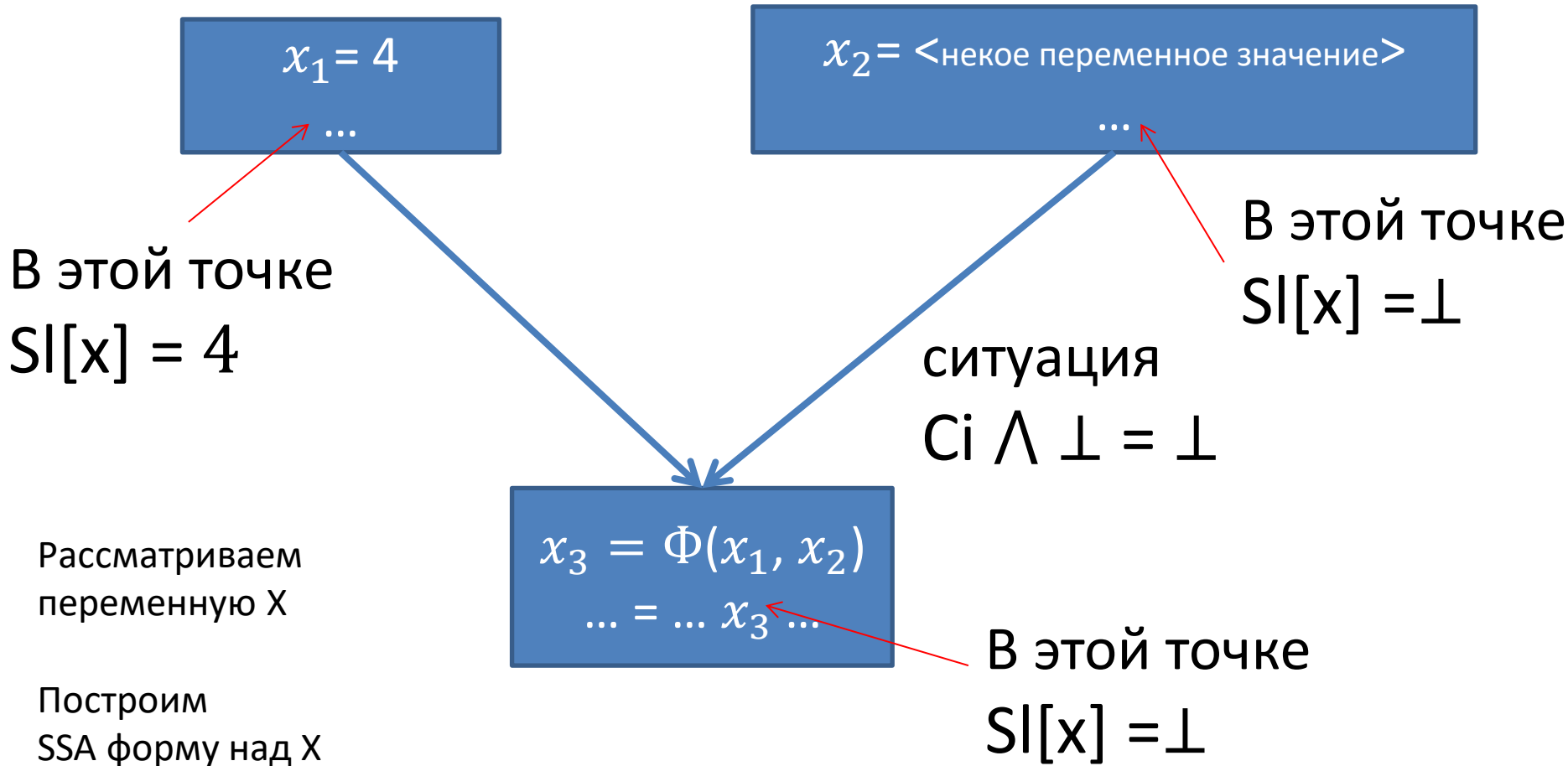




# Constant propagation



# Constant propagation



# Constant propagation

Полурешётки и выражения

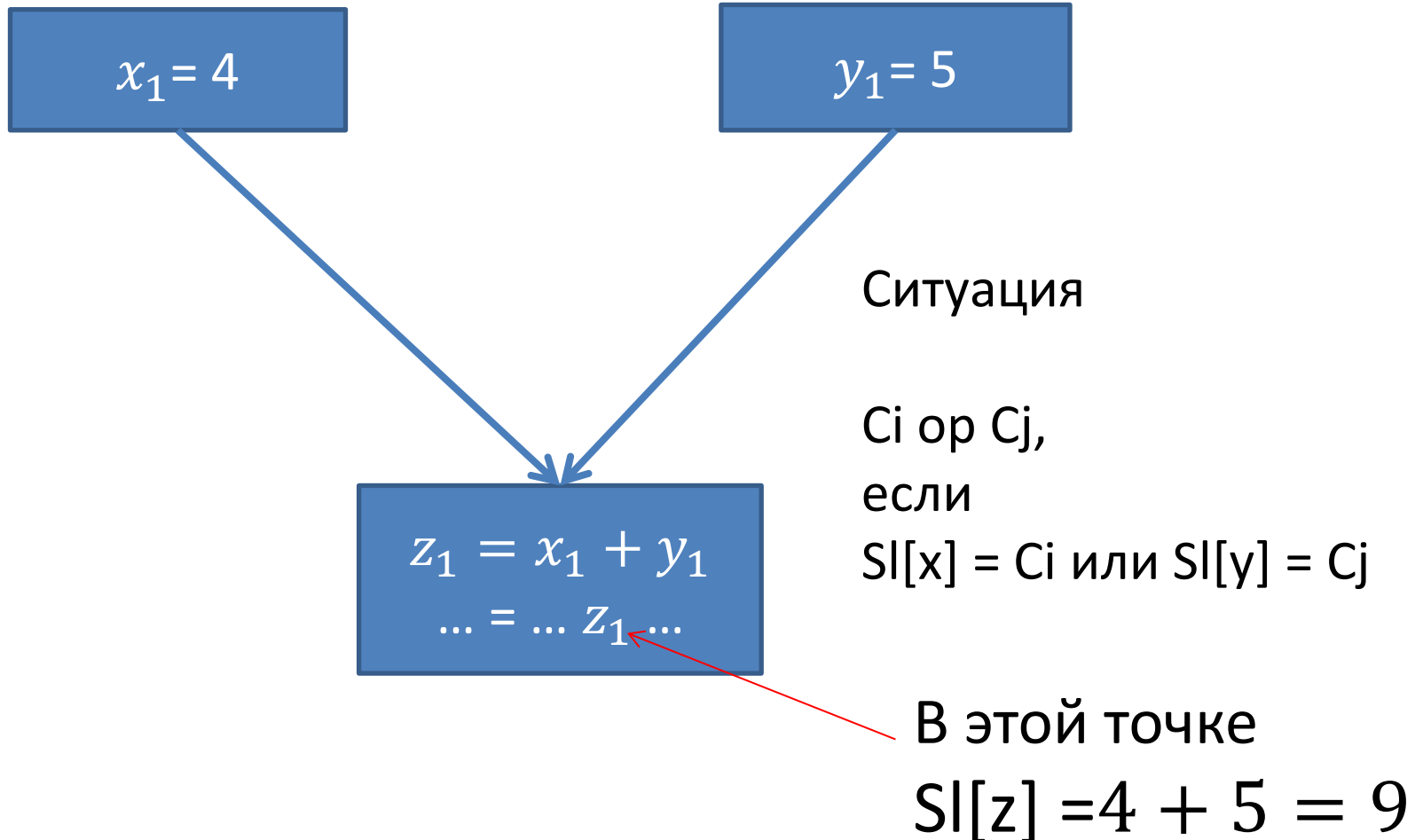
Пусть  $op$  – арифметический оператор (+, -, \*, /)

$SI[x \text{ op } y] =$

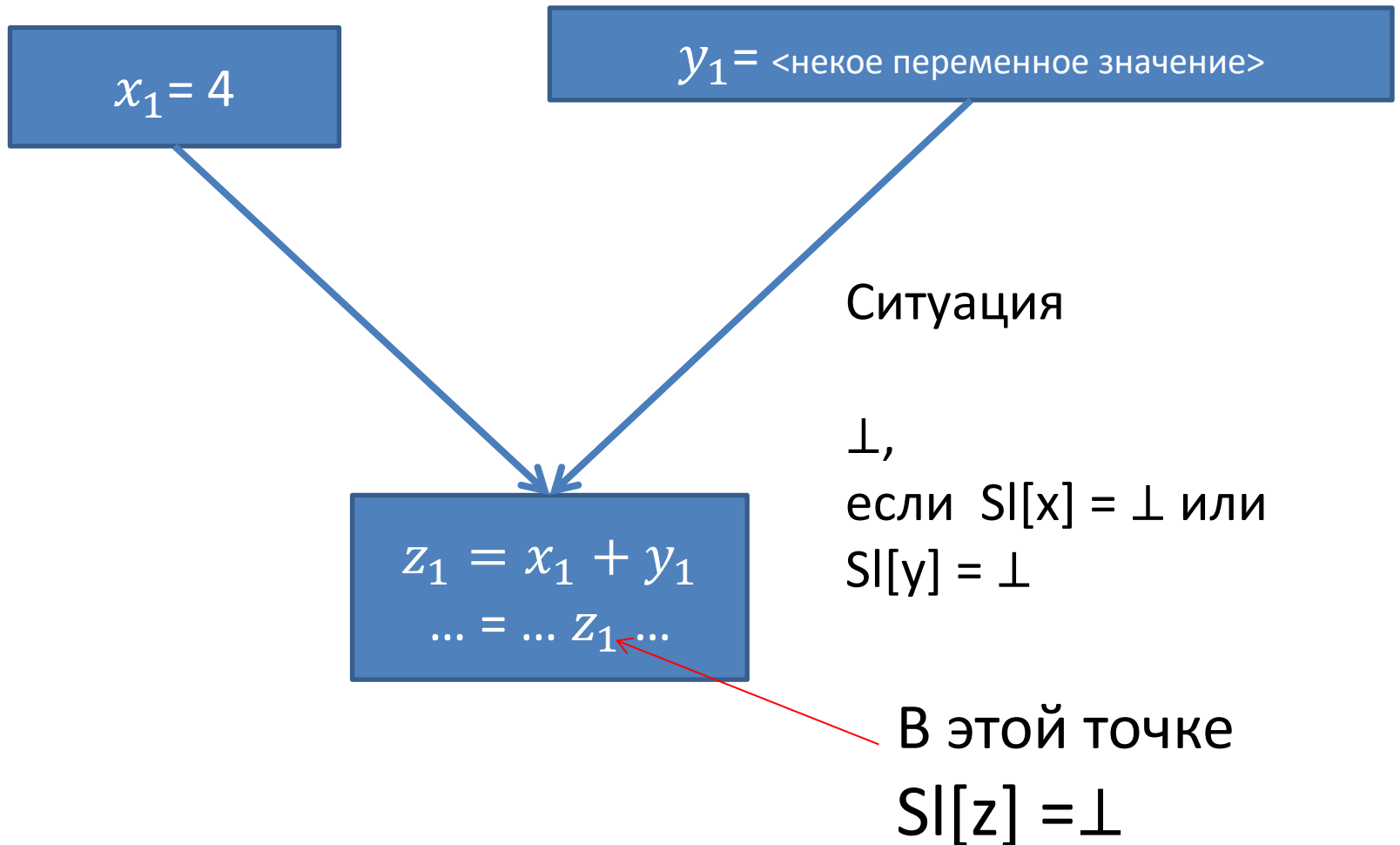
$\perp$ ,	если $SI[x] = \perp$ или $SI[y] = \perp$
$C_i \text{ op } C_j$ ,	если $SI[x] = C_i$ или $SI[y] = C_j$
$T$ ,	в остальных случаях

Будем использовать `LatEval( stmt s )`

# Constant propagation



# Constant propagation



# Constant propagation

Алгоритм распространения констант

Вход: CFG  $g$  с построенным SSA

ограничения:

- каждый ББ содержит одну инструкцию
- инструкции имеют вид “ $a = b \text{ op } c$ ” или “ $a = \text{op } b$ ”
- инструкции ветвления

Выход: выполненное преобразование

Метод:

set<edge> FlowWL;	// рабочий набор рёбер CFG
set<ssa_edge> SSAWL;	// рабочий набор рёбер SSA (def-use)
map<operand, p, sl> Sl;	// значения полурешётки
map<edge, <b>bool</b> > ExecFlag;	// true, если по ребру CFG передана
	// data-flow информация

```
Init(g);
```

```
while ( !FlowWL.is_empty() || !SSAWL.is_empty() )
```

```
{
```

```
    if ( !FlowWL.is_empty() )
```

```
    {
```

```
        edge e = FlowWL.del_item();
```

```
        if ( !ExecFlag[e] )
```

```
        {
```

```
            ExecFlag[e] = true;
```

```
            if ( e.v2.stmts[0].is_phi() )
```

```
                VisitPhi( e.v2.stmts[0] );
```

```
            else if ( EdgeCount( e.v2 ) == 1 )
```

```
                VisitInst(e.v2, e.v2.stmts[0] );
```

```
        }
```

```
    }
```

```
    if ( !SSAWL.is_empty() )
```

```
    {
```

```
        ssa_edge e = SSAWL.del_item();
```

```
        if ( e.v2.stmts[0].is_phi() )
```

```
            VisitPhi( e.v2.stmts[0] );
```


```
        else if ( EdgeCount( e.v2 ) >= 1 )
```

```
            VisitInst(e.v2, e.v2.stmts[0] );
```

```
    }
```

```
}
```

Выбрать любой  
элемент из мн-ва  
и удалить его




e.v1 – начальный  
ББ ребра

e.v2 – конечный  
ББ ребра

is\_phi() – true  
если это  
Ф-функция

Выбрать любой  
элемент из мн-ва  
и удалить его



```
void Init(CFG g)
```

```
{
```

```
    FlowWL.clear();
```

```
    “Добавить в FlowWL рёбро вида m->n, где m –  
входная вершина”;
```

```
    SSAWL.clear();
```

```
    foreach( edge e in AllEdge(g) )
```

```
        ExecFlag[e] = false;
```

```
    foreach( vertex v in AllVert(g) )
```

```
{
```

```
    stmt s in v.stmts[0];
```

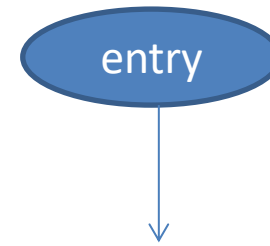
```
    SI[ s.lhs.var ] = “Т”; // точку p
```

```
                        // опустили
```

```
                        // для краткости
```

```
}
```

```
}
```



Пустой entry  
ББ

AllEdge – все  
рёбра CFG

AllVert – все ББ  
CFG

v.stmts –  
инструкции ББ



```
int EdgeCount( vertex v )
```

```
{
```

```
    int i = 0;
```

```
    foreach( edge e in InEdge(v) )
```

```
        if ( ExecFlag(e) )
```

```
            i++;
```

```
    return i;
```

```
}
```

InEdge – мн-во  
входящих дуг

```
void VisitPhi( stmt s )
```

```
{
```

```
    foreach( var v in s.rhs.vars )
```

```
    {
```

```
        SI[ s.lhs.var ] “∧=” SI[v];
```

```
    }
```

```
}
```

**void** VisitInst(vertex v, stmt s)

{

val = LatEval(s);

**if** ( val != SI[ s.lhs.var ] )

{

SI[ s.lhs.var] “ $\wedge$ =” val;

SSAWL += “SSASucc(v)”;

**if** ( val == “T” )

FlowWL += OutEdge(v);

**else if** ( val != “ $\perp$ ” )

{

**if** ( OutEdge(v).count() == 2 ) // случай инструкции ветвления и  
// условие константное

{

**foreach**(edge e in OutEdge(v) )

{

**if** (

(val && “e соответствует TRUE” ) ||

(!val && “e соответствует FALSE” )

)

FlowWL += e;

}

}

**else if** ( OutEdge(v).count() == 1 )

{

FlowWL += OutEdge(v);

}

}

}

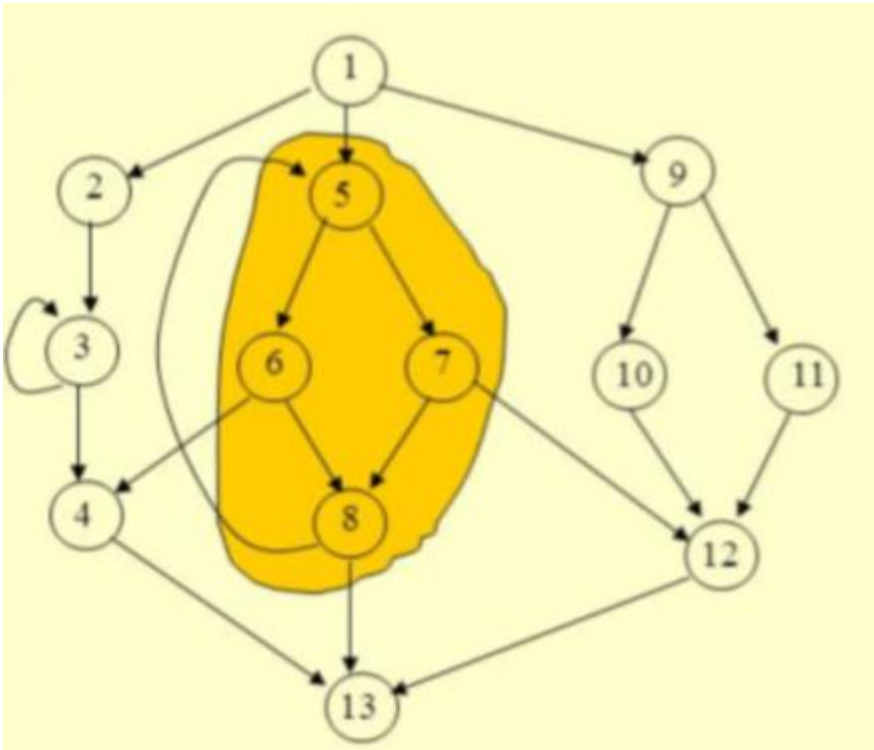
}

SSASucc(v) – мн-во исходящих  
def-use для инструкции в v

OutEdge(v) – мн-во исходящих  
рёбер CFG из v

# Dead code elimination

# Dominance Frontier (фронт доминаторов)



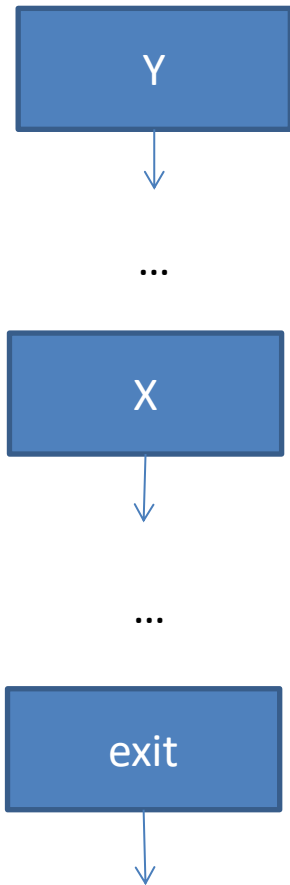
См. также фронт доминаторов в построение SSA (вставка PHI функций)

Узлы 4, 13, 12 представляют собой блоки, над которыми блок 5 не доминирует и которые встречаются первыми на путях из 5. Т.е. граница, где исчезает доминирование блока 5.

**$DF(5) = \{ 4, 5, 12, 13 \}$**

Фронт постдоминаторов рассматривается аналогично.

# PostDominator



Def:  $X \text{ pdom } Y$

Все пути от  $Y$  до exit (выходной вершины) проходят через  $X$

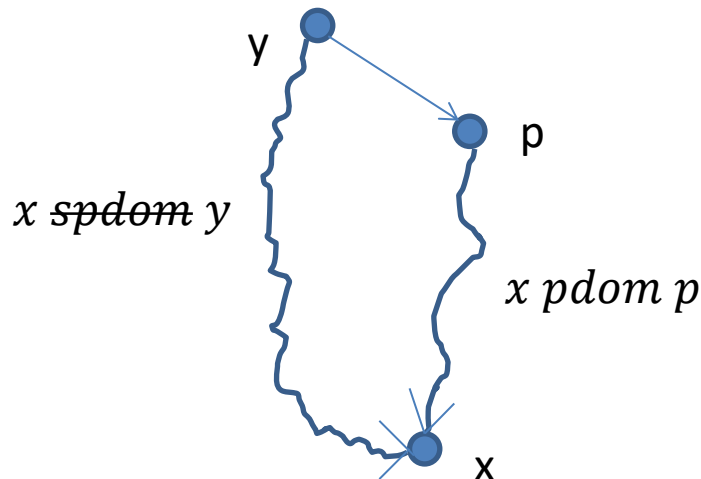
# PostDominance Frontier (фронт постдоминаторов)

Def: PostDominance frontier  $PostDF(x)$  для одной вершины

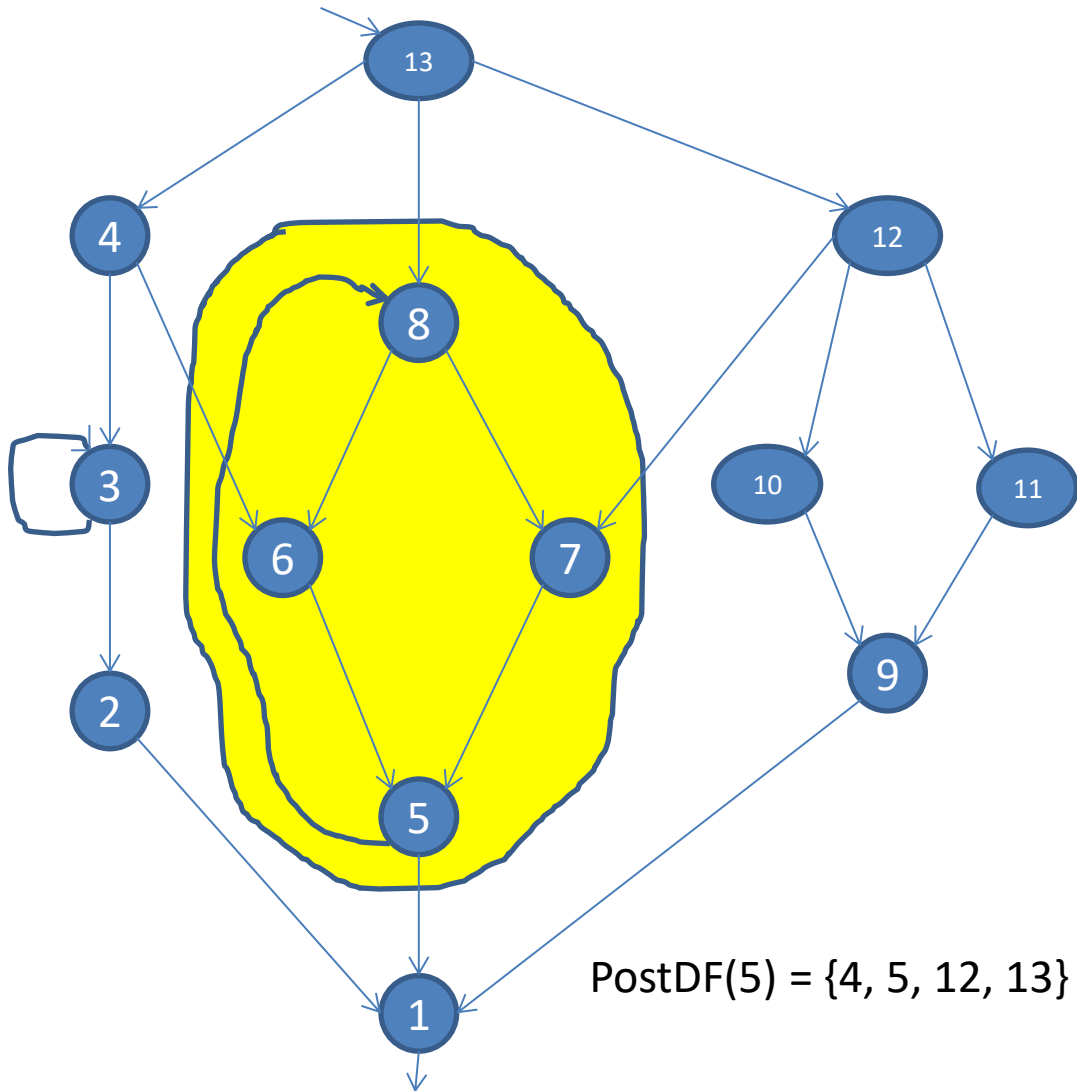
Пусть дан некий CFG

$x$  – вершина CFG графа

$$PostDF(x) = \{y \mid \exists p \in Succ(y) \\ x \text{ pdom } p \text{ \&\& } x \text{ ~~spdom~~ } y\}$$



# PostDominance Frontier (фронт постдоминаторов)



Узлы 4, 13, 12 представляют собой последние блоки при входе в область постдоминирования 5

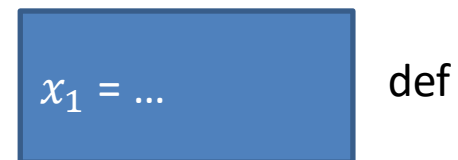
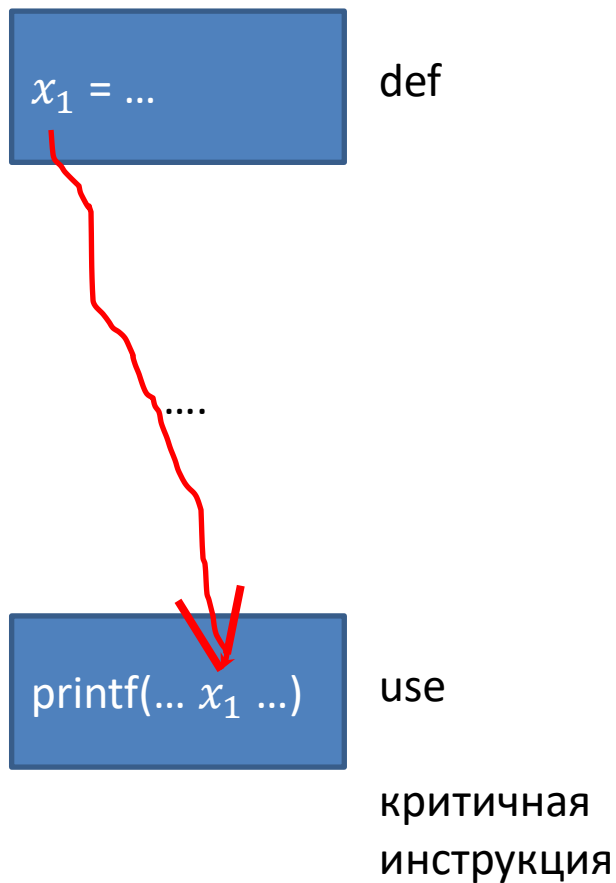
$$\text{PostDF}(5) = \{4, 5, 12, 13\}$$

# Dead Code Elimination

- Критичные инструкции
  - I/O операции
  - return
  - Вызовы других функций
  - т.д.
- Mark / Sweep алгоритм

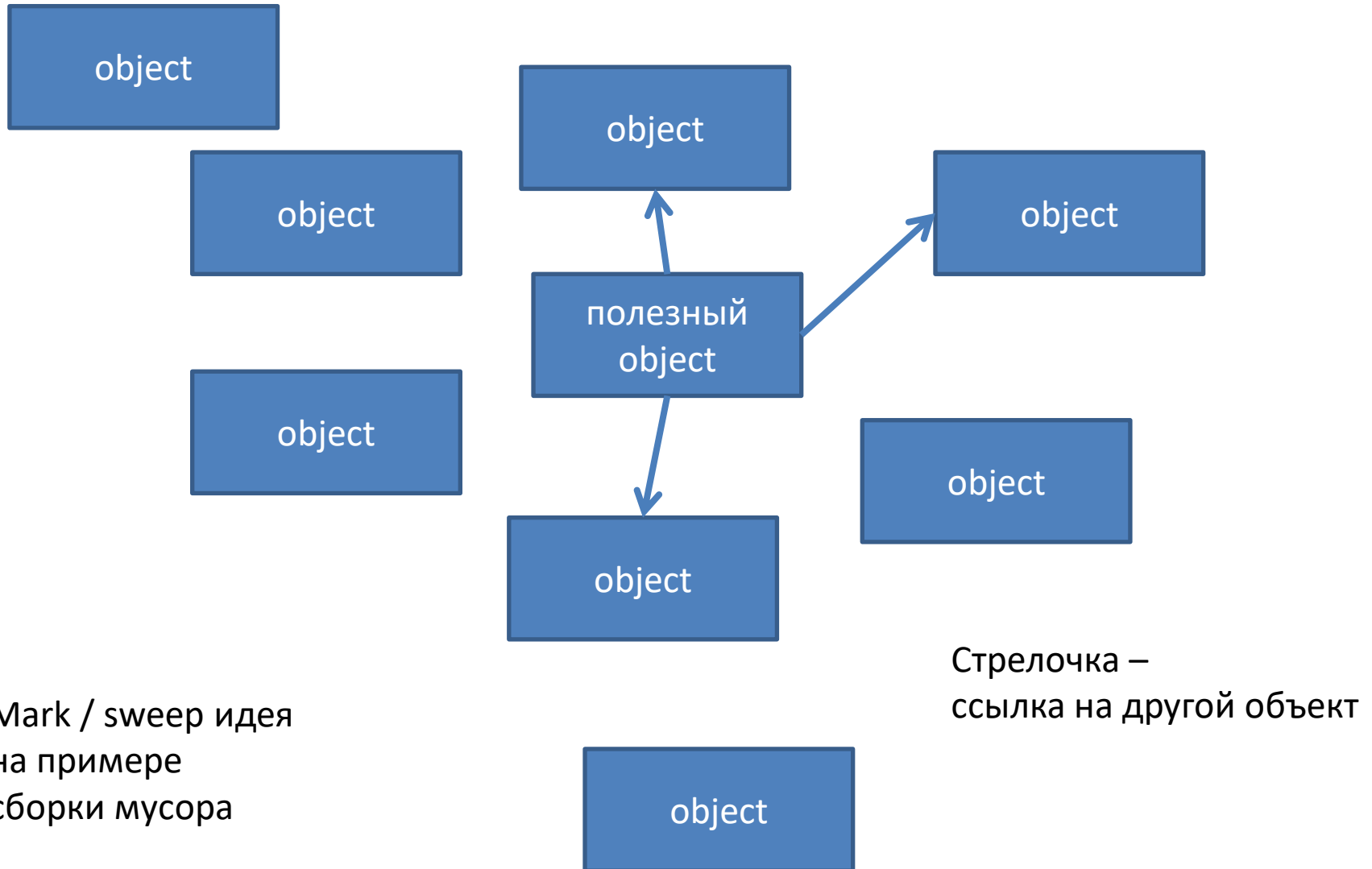


# Dead Code Elimination

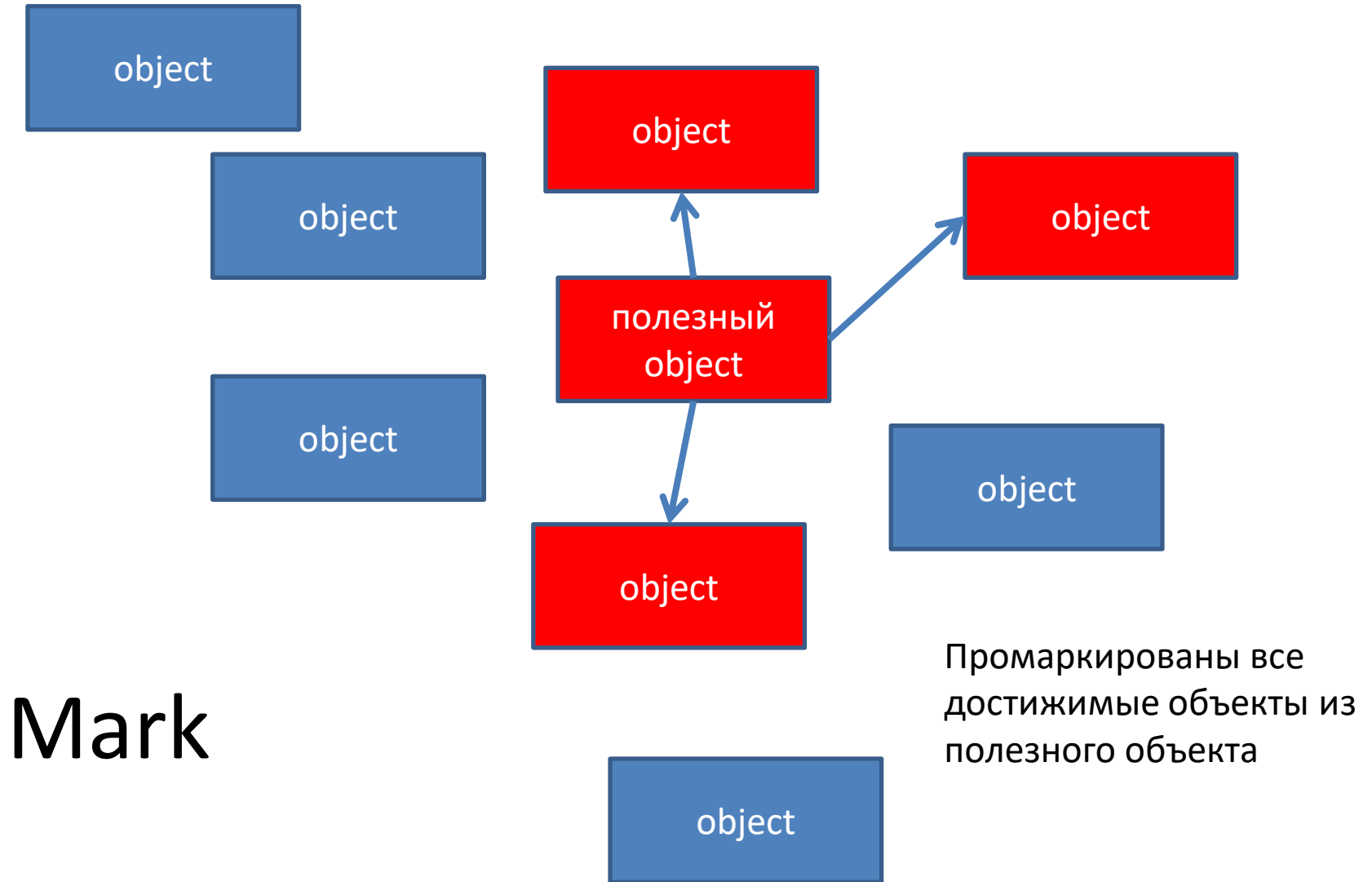


Использовать def-use линки

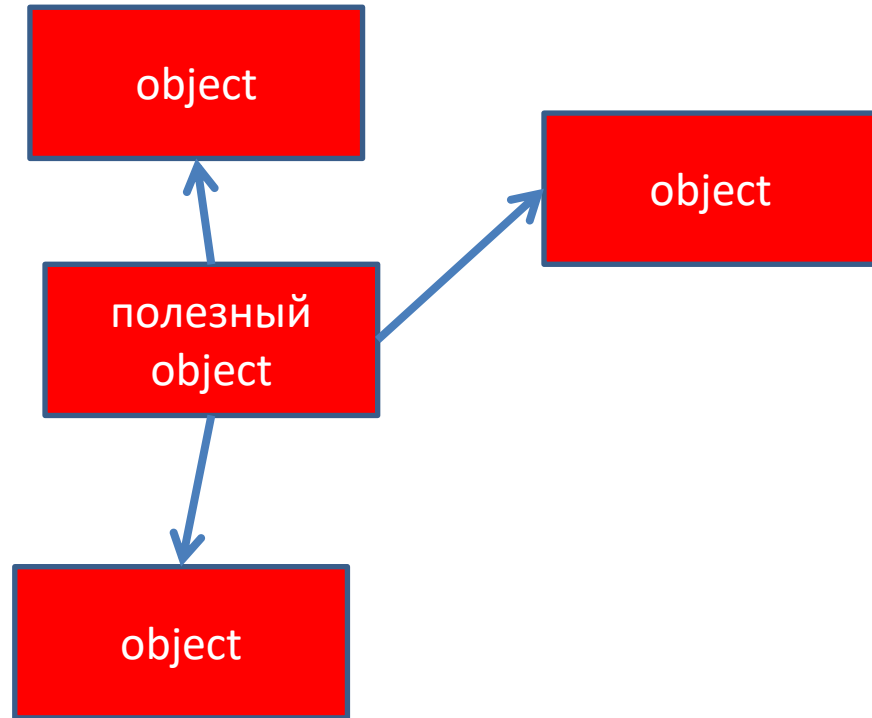
# Dead Code Elimination



# Dead Code Elimination



# Dead Code Elimination



Sweep

Удаляем недостижимые  
объекты

# Dead Code Elimination

Алгоритм удаления мёртвых инструкций

Вход: CFG  $g$  с построенным SSA, Pdom деревом, PostDF для каждой вершины

Выход: выполненное преобразование

Метод:

`set<stmt> WorkList; // изначально пустое`

`set<stmt> Mark; // изначально пустое`


# Dead Code Elimination

```
// Mark
for “each stmt s in CFG”
{
    if s “is critical”
    {
        Mark.add(s);
        WorkList.add(s);
    }
}
```

# Dead Code Elimination

```
while (!WorkList.is_empty())  
{
```

Выбрать любой  
элемент из мн-ва  
и удалить его



```
    stmt s = WorkList.del_item();
```

“для всех аргументов s берём stmt def” // используем SSA,  
// т.е. идём от use к def

```
{  
    Mark.add(def);  
    WorkList.add(def);  
}
```

```
foreach Vertex v in PostDF(s.vertex) // рисунок #1
```

```
{  
    stmt branch = v.last; // берём инструкцию ветвления в базовом блоке v  
  
    Mark.add(branch);  
    WorkList.add(branch);  
}  
}
```

# Dead Code Elimination

// Sweep

for “each stmt s in CFG”

{

if ( ! Mark.in(s) )

{

if (s.is\_branch()) // рисунок #2

“перенаправляем на ближайший полезный

Post-доминатор”;

else

“удаляем s из CFG”;

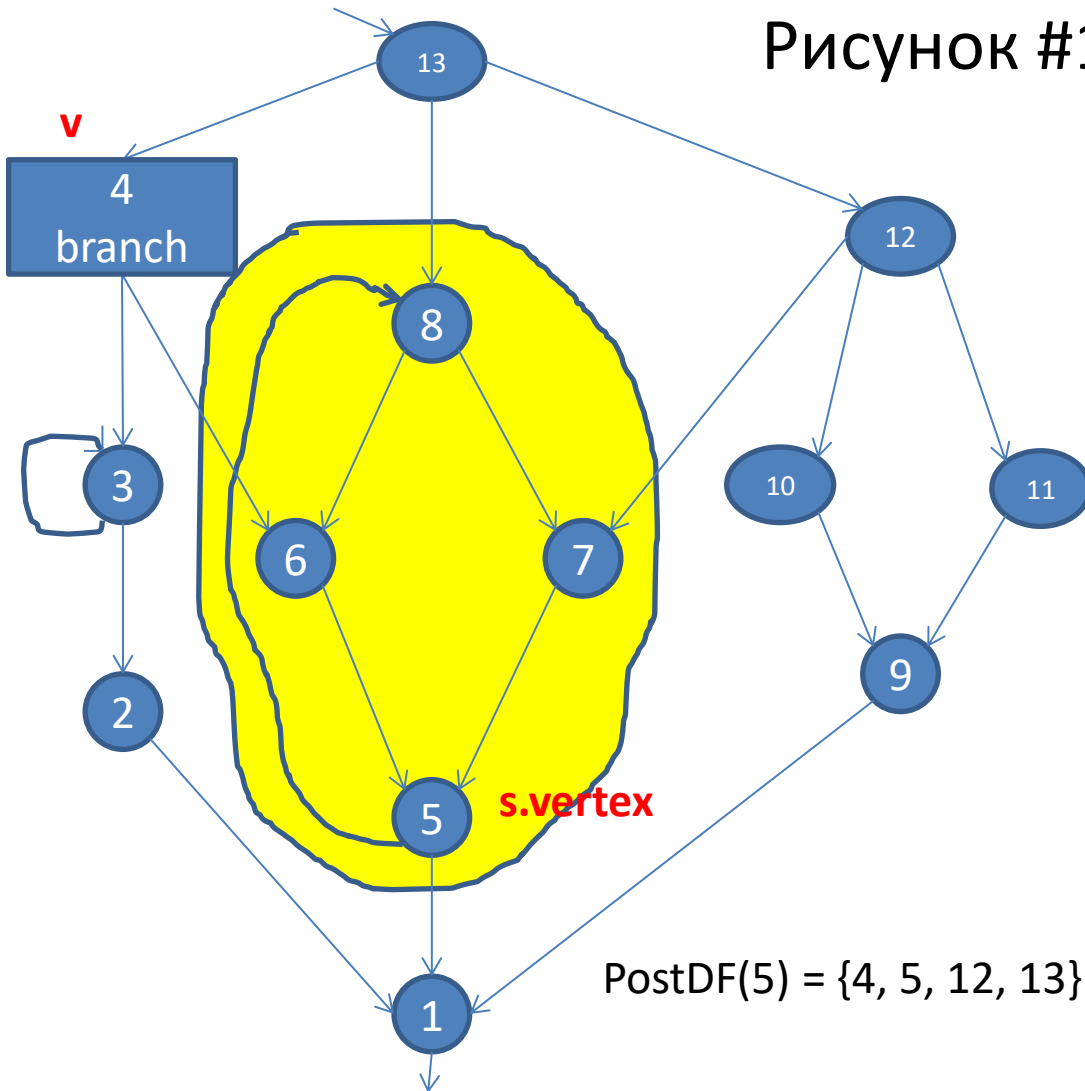
}

}



# Dead Code Elimination

Рисунок #1



s.vertex содержит  
значимый (“живой”) код

Факт выполнения s.vertex  
зависит от выполнения  
инструкции branch

=>

branch – значимая (“живая”)  
инструкция

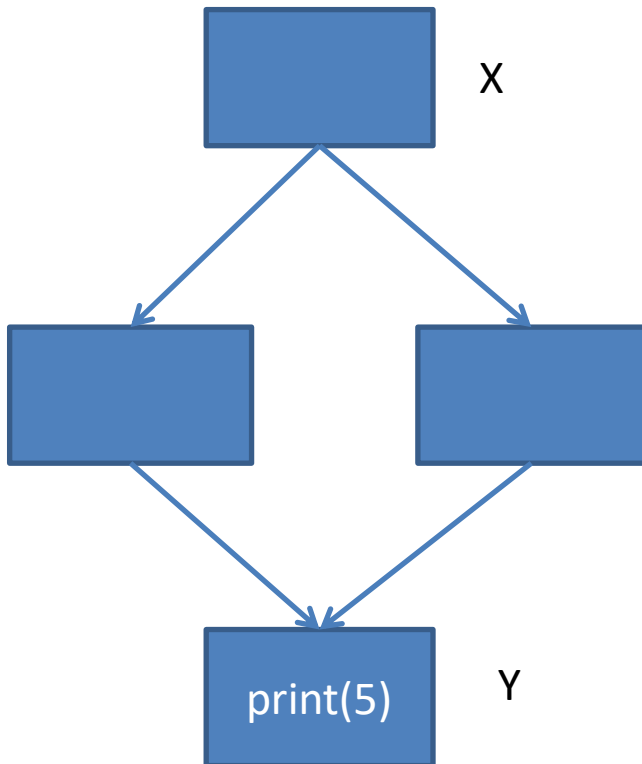
=>

$\text{PostDF}(5) = \{4, 5, 12, 13\}$

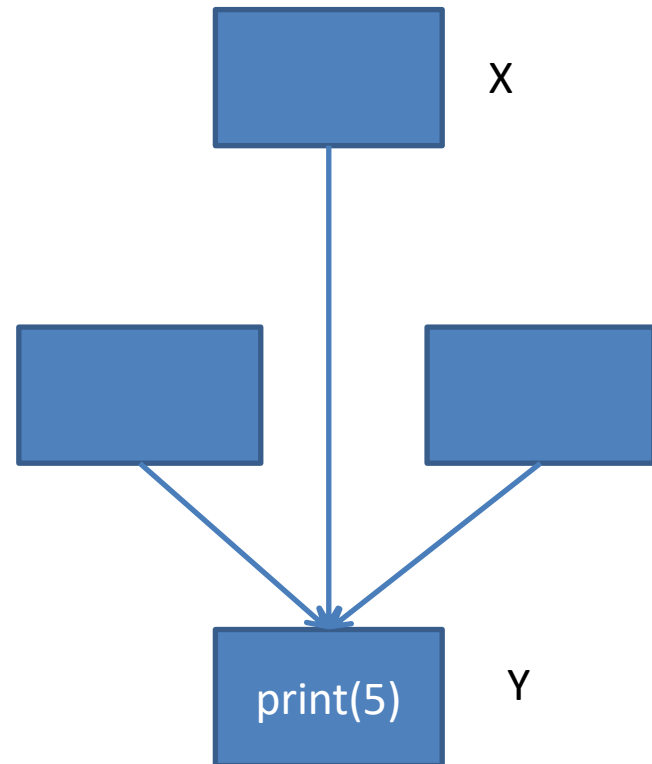
```
Mark.add(branch);  
WorkList.add(branch);
```

# Dead Code Elimination

Рисунок #2



Везде код мёртвый за  
исключением print(5)



Y – ближайший постдоминатор для X

# Список литературы

- Advanced compiler design & implementation.  
S. Muchnik (все оптимизации)