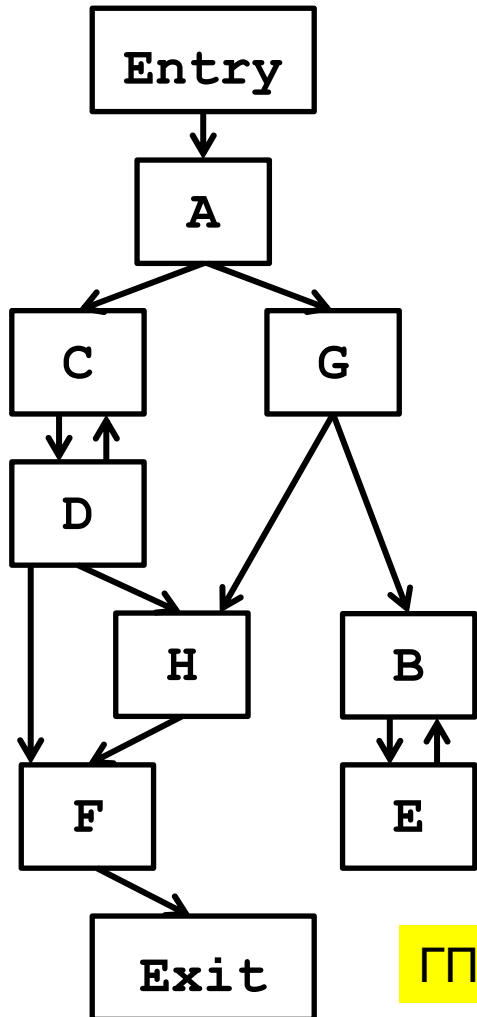


2. Построение множеств Input и Output

2.1 Нумерация вершин ГПУ

2.1.1 Остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию»



ГПУ

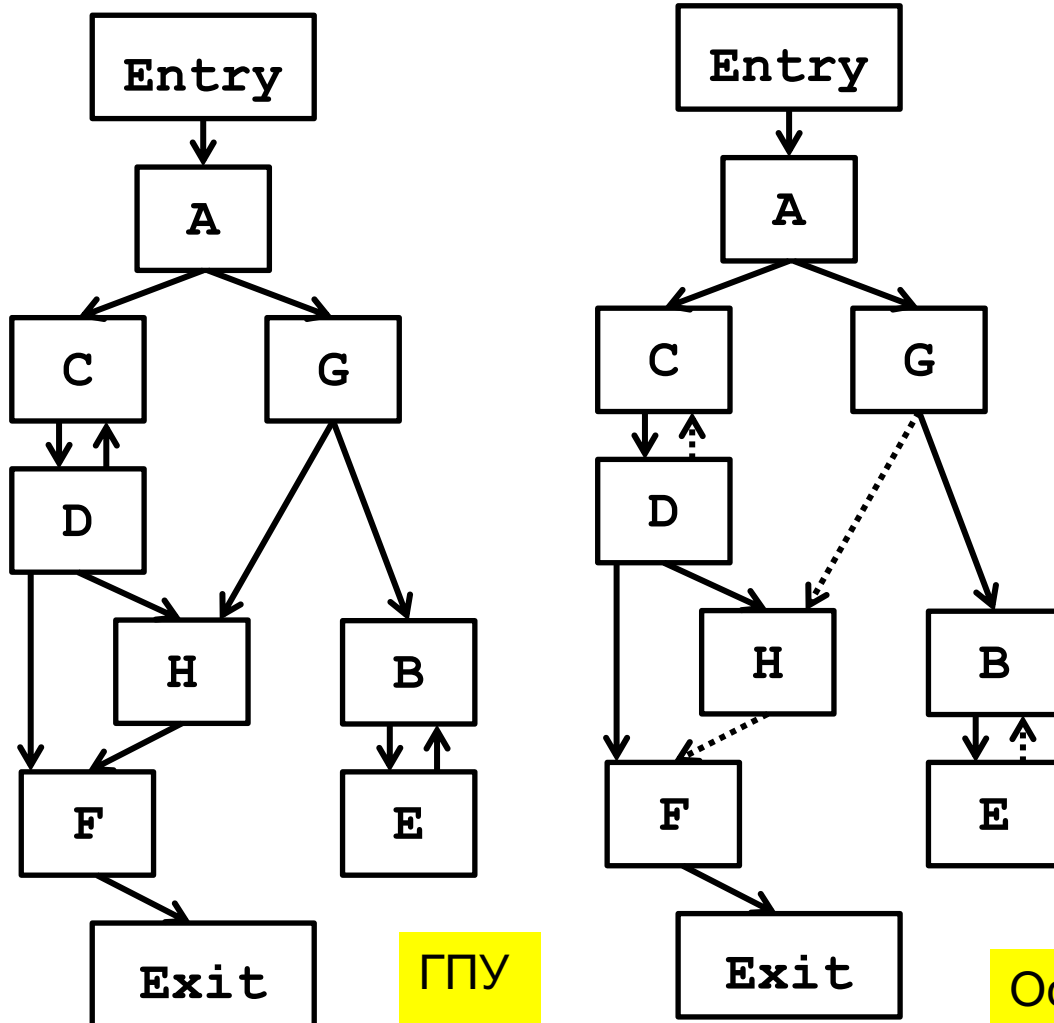
Остовное дерево графа содержит **все** вершины графа и часть его дуг.

Обратная нумерация используется для того, чтобы, например, вершина А имела номер 1, а не 8.

2.1 Нумерация вершин ГПУ

2.1.2 Глубинное остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию»

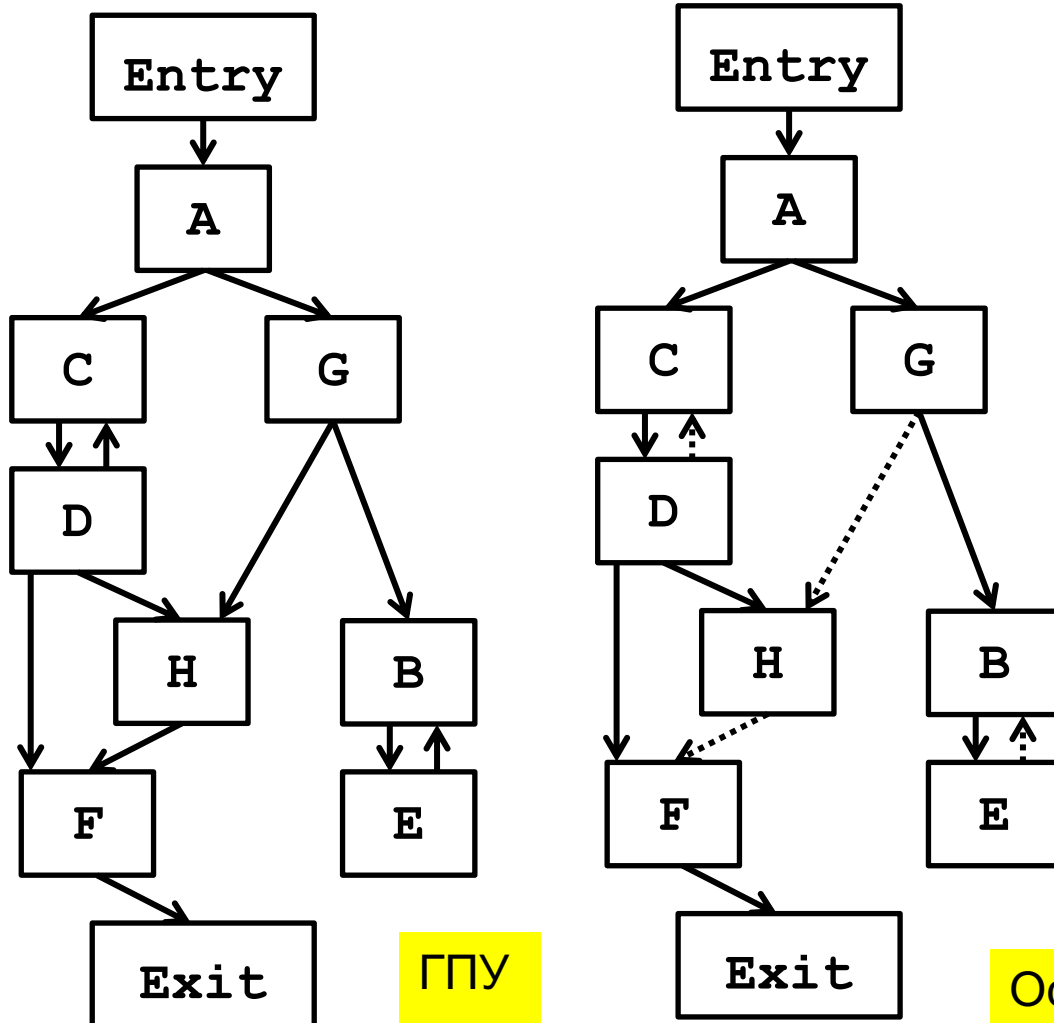


«Лишние» дуги ГПУ, не вошедшие в ОД, на рисунке изображены пунктиром

2.1 Нумерация вершин ГПУ

2.1.2 Глубинное остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем это дерево слева направо «сначала в глубину», используя «обратную нумерацию»



«Лишние» дуги ГПУ, не вошедшие в ОД, на рисунке изображены пунктиром

Такое остовное дерево будем называть глубинным остовным деревом, чтобы помнить, как пронумерованы его вершины

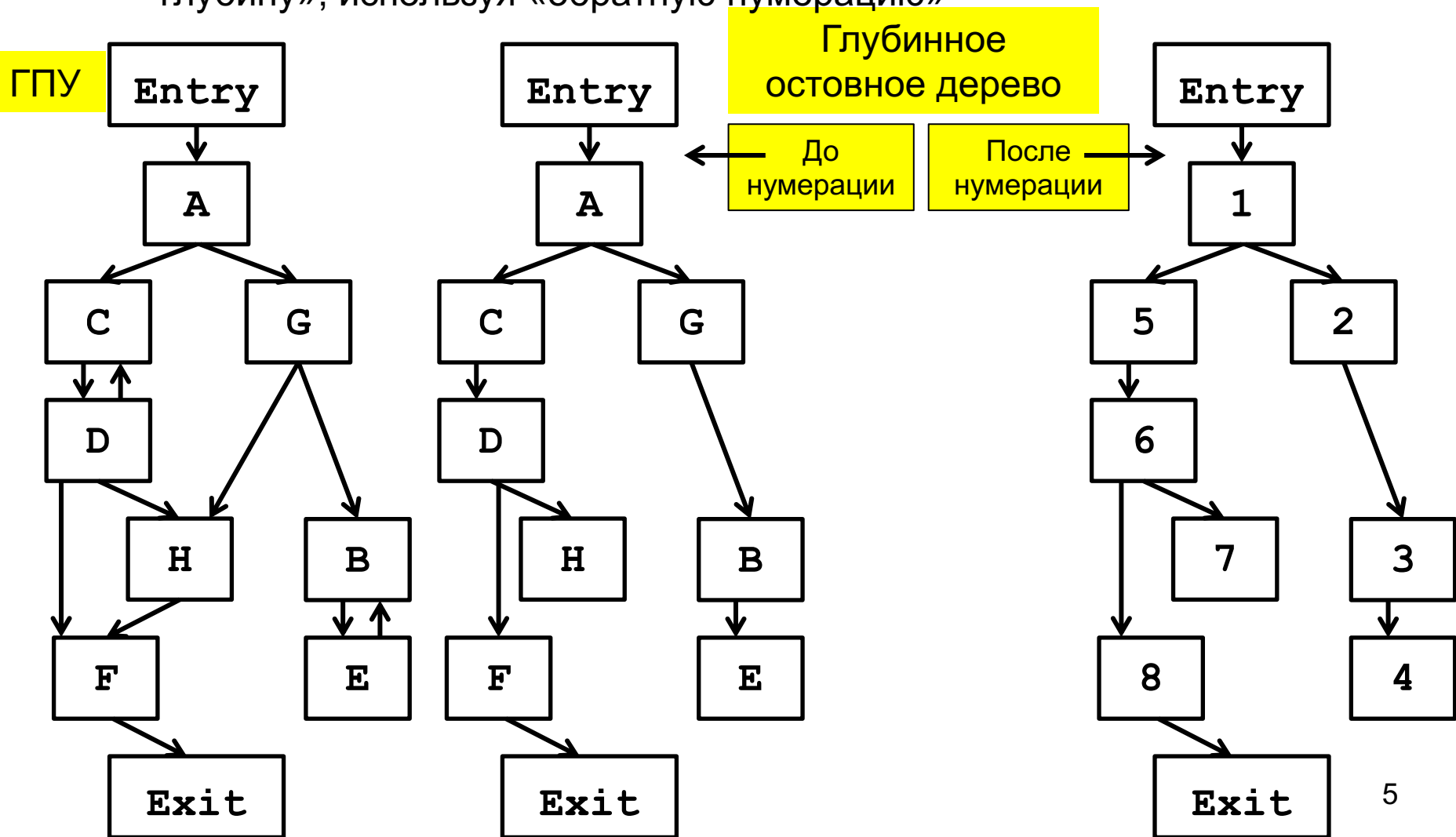
ГПУ

Остовное дерево

2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

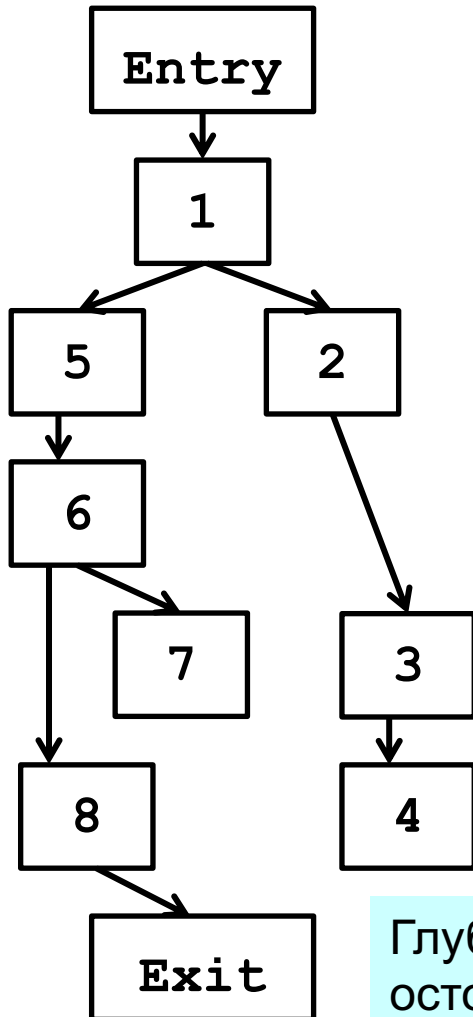
- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем его слева направо «сначала в глубину», используя «обратную нумерацию»



2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем его слева направо «сначала в глубину», используя «обратную нумерацию»



Глубинное
остовное дерево

В случае обратной нумерации вершин графа, содержащего n вершин i -ой вершине присваивается номер $n - i$

На остовном дереве идентификаторы вершин заменяем их номерами

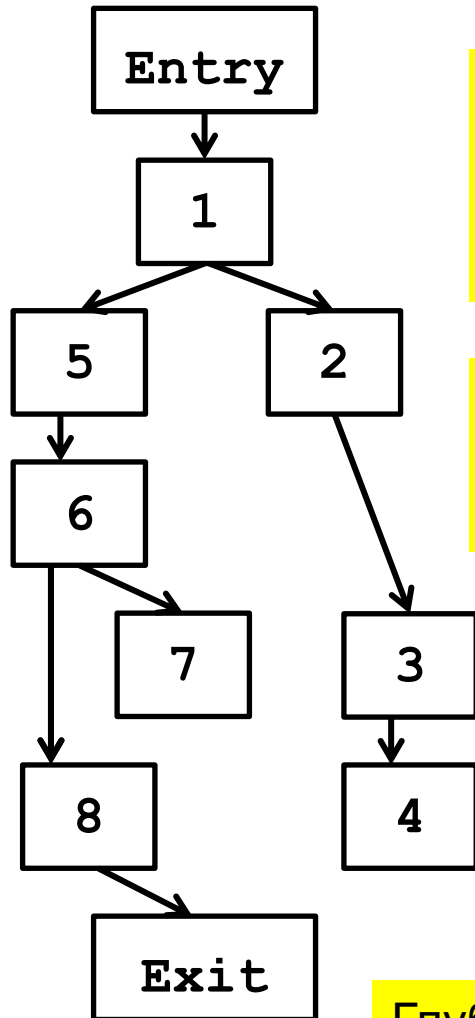
Остовное дерево с корнем в **Entry** и такой нумерацией вершин называется *глубинным остовным деревом* – *DFST*).

DFST – *Depth First Spanning Tree*

2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

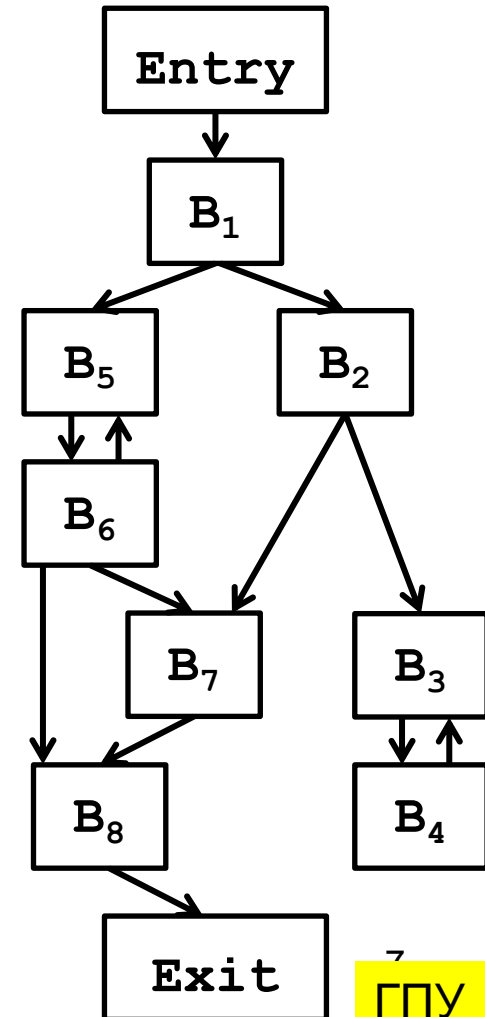
- ◇ Чтобы пронумеровать вершины ГПУ, построим его *остовное дерево* (ОД) с корнем в вершине **Entry** и обойдем его слева направо «сначала в глубину», используя «обратную нумерацию»



Присвоив номера вершин остовного дерева соответствующим базовым блокам, получим ГПУ с пронумерованными вершинами.

Нумерация вершин ГПУ определяет порядок их посещения во время обходов ГПУ

Глубинное остовное дерево

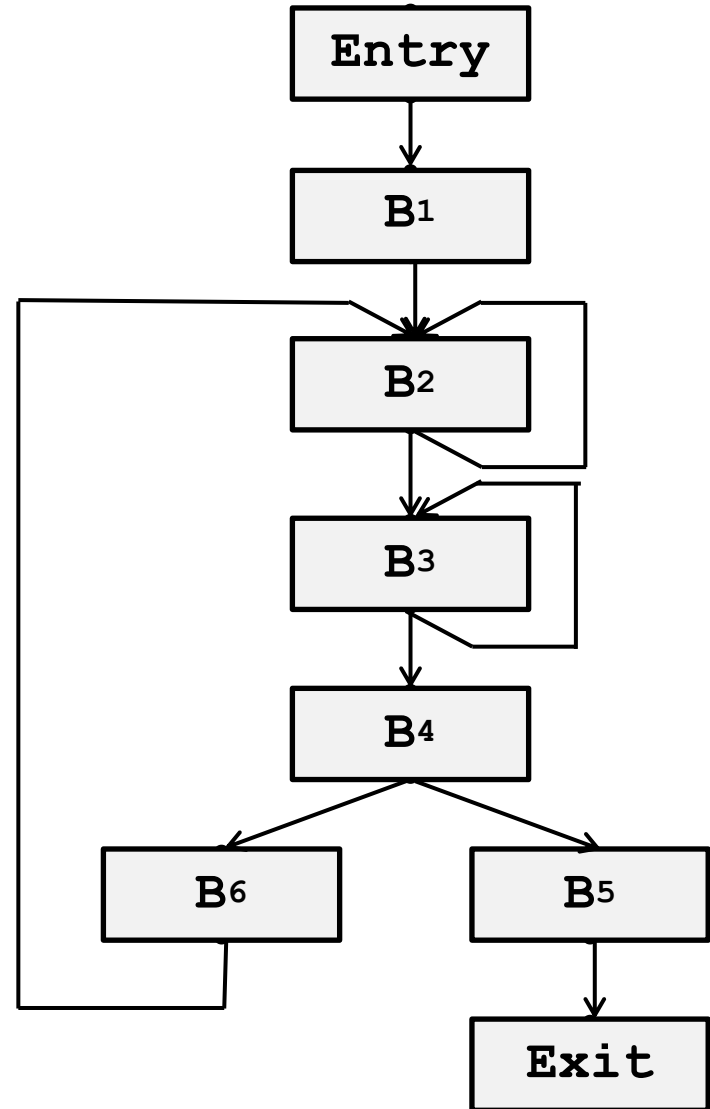
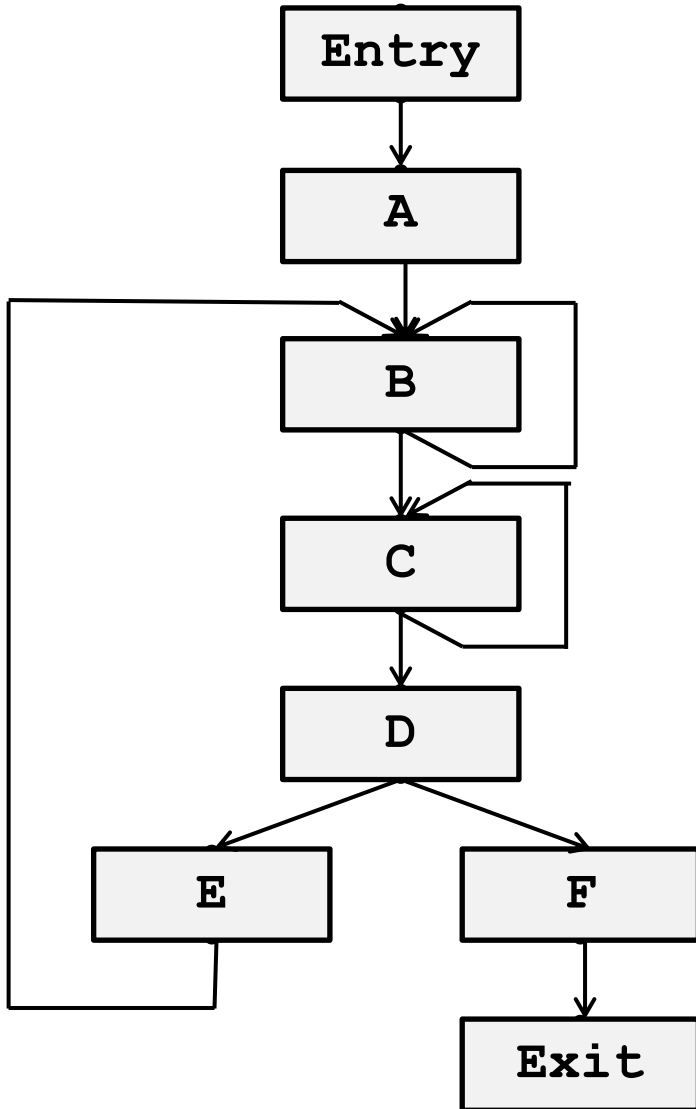


ГПУ

2.1 Нумерация вершин ГПУ

2.1.1 Глубинное остовное дерево

◇ После нумерации вершин ГПУ из примера 1.5.5 (тема 1) примет вид



2.1 Нумерация вершин ГПУ

2.1.2 Алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ

Алгоритм

- ◇ **Вход:** ГПУ $G = \langle N, E \rangle$ с корнем $Entry \in N$
- ◇ **Выход:** глубинное остовное дерево графа G ($T_{DFS}(G)$) и нумерация узлов графа G , соответствующая упорядочению в глубину.
- ◇ **Метод:** все узлы $n \in N$ помечаются как nv (*not visited*)
вызывается рекурсивная процедура **DFST (n0)**
когда процедура **DFST** завершится, будут построены:
 - ◇ массив узлов dfn в порядке новой нумерации
 - ◇ множество T ребер глубинного остовного дерева $T_{DFS}(G)$

2.1 Нумерация вершин ГПУ

2.1.2 Алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ

Рекурсивный алгоритм построения *DFST*

◇ Функция `main()` :

```
main() {  
    T = ∅;  
    for all n ∈ N n.vst = nv;  
    c = |N|; // |N| = кол-во узлов  
    DFST(n0);  
}
```

Каждая вершина ГПУ представлена структурой

```
struct n {number, vst},
```

где `number` – номер вершины, а

`vst` имеет 2 значения:

`v` (вершина была посещена) и

`nv` (вершина не была посещена)

2.1 Нумерация вершин ГПУ

2.1.2 Алгоритм построения глубинного остовного дерева и нумерации вершин ГПУ

◇ Функция `DFST()`:

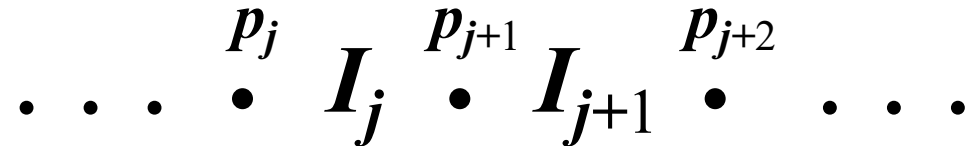
```
void DFST(n) {
    Отмечаем n как v;
    for all s ∈ Succ(n)
        if (s.vst == nv) {
            T ∪= {n→s};
            DFST(s);
        }
    // узлу n соответствует номер c
    n.number = c;
    dfn[c] = n;
    c--;
}
```

◇ **Замечание.** Для каждой вершины $n \in N$ нетрудно построить множество $Succ(n)$, содержащее все вершины $s \in N$, в которые входят дуги, выходящие из вершины n .

2.2 Анализ потока данных

2.2.1 Поток данных

- ◇ *Состояние программы* – множество значений всех переменных программы, включая переменные в кадрах стека времени выполнения, находящихся ниже текущей вершины стека
- ◇ *Точки программы* $(\dots, p_j, p_{j+1}, p_{j+2}, \dots)$ расположены между ее инструкциями $(\dots, I_j, I_{j+1}, \dots)$



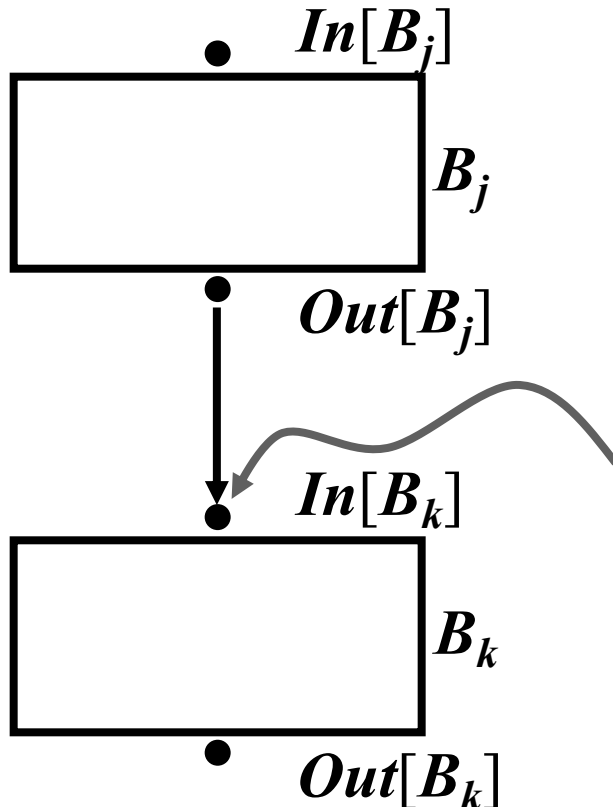
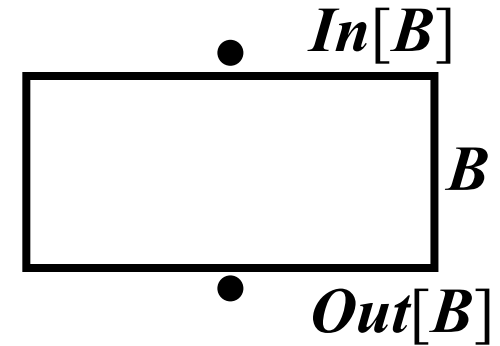
- ◇ Инструкция программы I_j описывается парой состояний:
 - ◇ состоянием в *точке программы* p_j *перед* инструкцией I_j
 - ◇ состоянием в *точке программы* p_{j+1} *после* инструкции I_j .

2.2 Анализ потока данных

2.2.1 Поток данных

Базовый блок B описывается парой состояний:

- ◇ состоянием $In[B]$ в точке входа в B (перед первой инструкцией),
- ◇ состоянием $Out[B]$ в точке выхода из B (после последней инструкции)



С дугой от блока B_j к блоку B_k связаны две точки программы:

- ◇ точка выхода из блока B_j (ей соответствует состояние $Out[B_j]$)
- ◇ точка входа в блок B_k (ей соответствует состояние $In[B_k]$)

При рассмотрении потока данных между базовыми блоками нельзя отождествлять точку выхода из базового блока и точку входа в следующий за ним базовый блок, так как последняя может следовать за точками выхода из нескольких базовых блоков (есть же **goto**)

2.2 Анализ потока данных

2.2.3 Передаточные функции инструкций

- ◇ Соотношение f_{I_j} между значениями данных до и после инструкции I_j называется *передаточной функцией* инструкции I_j .
- ◇ Передаточные функции работают **в прямом** и **обратном** направлениях:
 - ◇ **В задаче прямого обхода:** $Out[I_j] = f_{I_j}(In[I_j])$
 - ◇ **В задаче обратного обхода:** $In[I_j] = f_{I_j}^b(Out[I_j])$
- ◇ Если I_j и I_{j+1} – *последовательные* инструкции блока B , то
 - ◇ **В задаче прямого обхода:** $In[I_{j+1}] = Out[I_j]$
 - ◇ **В задаче обратного обхода:** $Out[I_{j-1}] = In[I_j]$

2.2 Анализ потока данных

2.2.3 Передаточные функции инструкций

- ◇ Соотношение f_{I_j} между значениями данных до и после инструкции I_j называется *передаточной функцией* инструкции I_j .
- ◇ Передаточные функции работают в **прямом** и **обратном** направлениях:
 - ◇ В задаче прямого обхода: $Out[I_j] = f_{I_j}(In[I_j])$
 - ◇ В задаче обратного обхода: $In[I_j] = f_{I_j}^b(Out[I_j])$
- ◇ Если I_j и I_{j+1} – *последовательные* инструкции блока B , то
 - ◇ В задаче прямого обхода: $In[I_{j+1}] = Out[I_j]$
 - ◇ В задаче обратного обхода: $Out[I_{j-1}] = In[I_j]$

f и f^b – две разные функции
(для разных задач анализа
ПОТОКОВ ДАННЫХ)

2.2 Анализ потока данных

2.2.3 Передаточные функции базовых блоков

- ◇ Рассмотрим базовый блок

$$B = \langle P, Input, Output \rangle,$$

где $P = I_1, \dots, I_n$ (в указанном порядке)

- ◇ По определению

$$In[B] = In[I_1], Out[B] = Out[I_n].$$

- ◇ Передаточная функция f_B блока B по определению равна композиции передаточных функций его инструкций I_1, \dots, I_n

$$f_B(x) = f_{I_n}(f_{I_{n-1}}(\dots f_{I_1}(x)\dots)) = (f_{I_1} \circ f_{I_2} \circ \dots \circ f_{I_n})(x)$$

или

$$f_B = f_{I_n} \circ f_{I_{n-1}} \circ \dots \circ f_{I_1}$$

2.2 Анализ потока данных

2.2.4 Передаточные функции базовых блоков

◇ При прямом обходе:

Соотношение между потоком данных при выходе из блока B и потоком данных при входе в него имеет вид

$$Out[B] = f_B(In[B])$$

◇ При обратном обходе:

Соотношение между потоком данных при входе в блок B и потоком данных при выходе из него имеет вид

$$In[B] = f_B^b(Out[B])$$

2.3 Достигающие определения

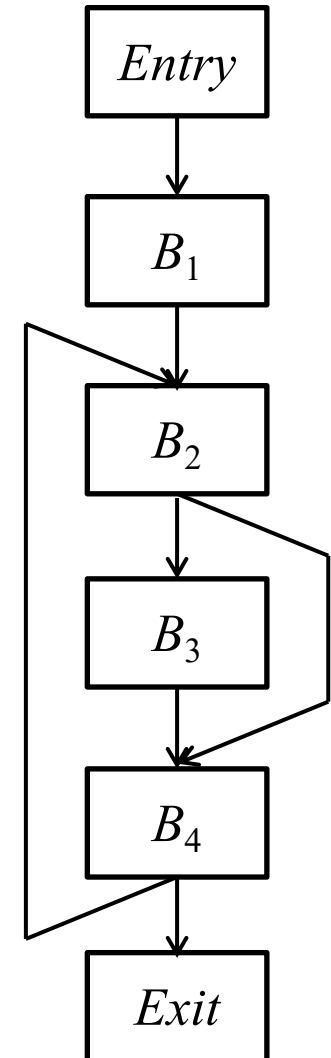
2.3.1 Терминология

- ◇ *Определением переменной x* называется инструкция, которая присваивает значение переменной x .
 - ◇ *Использованием переменной x* является инструкция, одним из операндов которой является переменная x .
 - ◇ Каждое определение переменной x *убивает* все другие ее определения.
 - ◇ Определение d *достигает* точки p , если существует путь от точки, непосредственно следующей за d , к точке p , такой, что вдоль этого пути d остается живым.
- ◇ **Замечание.** Во время анализа достигающих определений рассматриваются не переменные, а их определения, причем каждая переменная может иметь несколько определений

2.3 Достигающие определения

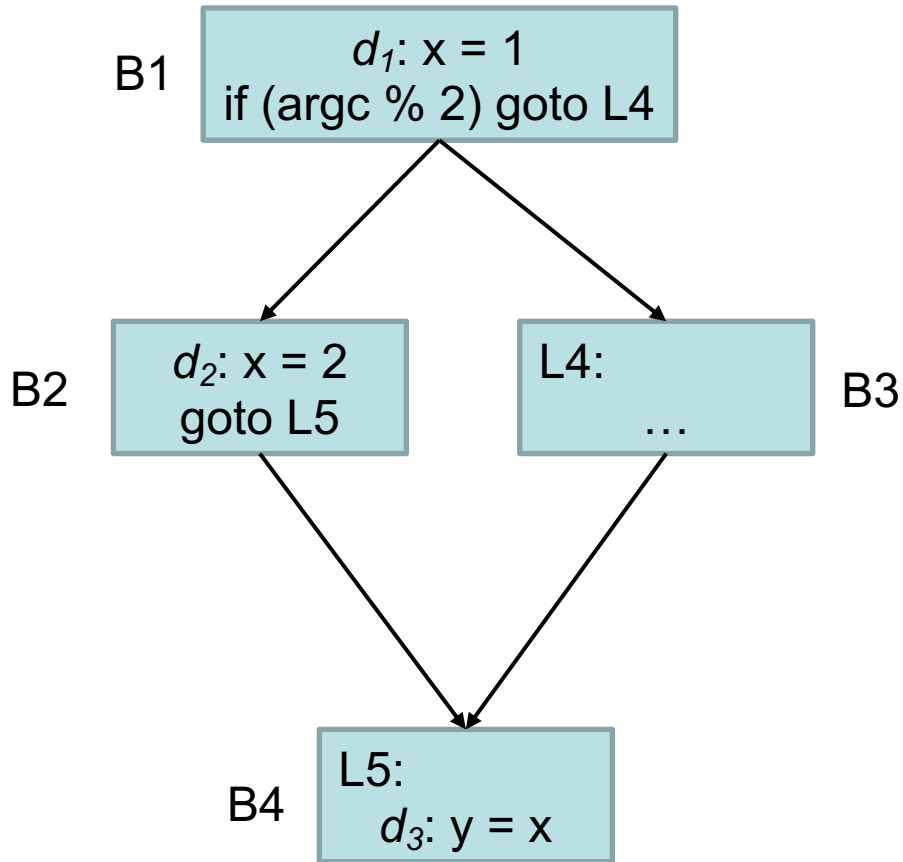
2.3.2 Пример

B_1 $i \leftarrow -, m, 1$ $j \leftarrow n$ $a \leftarrow u1$	B_2 $i \leftarrow +, i, 1$ $j \leftarrow -, j, 1$
B_3 $a \leftarrow u2$	B_4 $i \leftarrow u3$



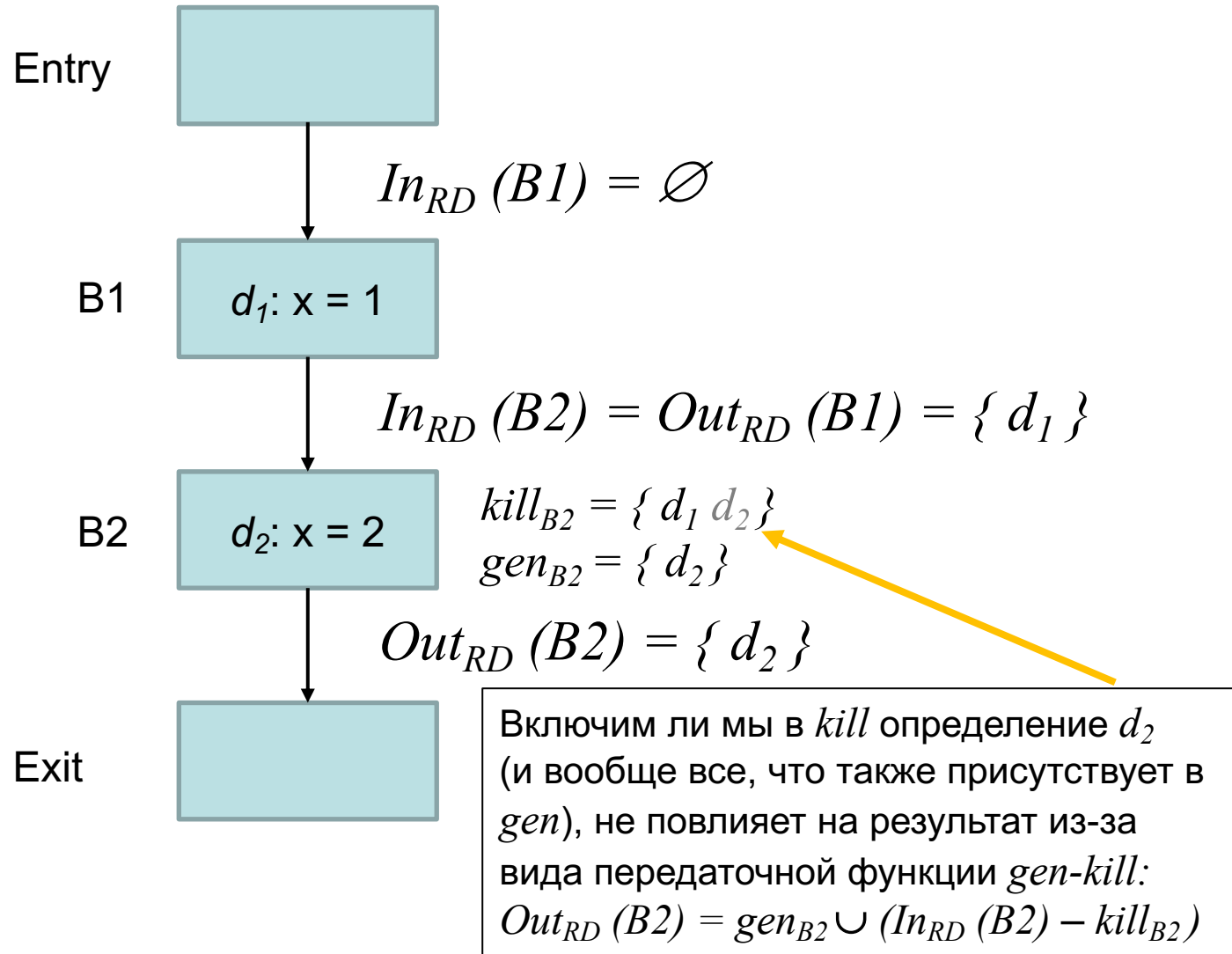
- ◇ Начало блока B_2 **достигается** определениями:
 - ◇ $(i, B_1), (j, B_1), (a, B_1),$
 - ◇ (j, B_2) (других определений j на пути от (j, B_2) до начала блока B_2 нет)
 - ◇ (a, B_3)
 - ◇ (i, B_4)
- ◇ Начало блока B_2 **не достигается** определением: (i, B_2) , так как его убивает определение (i, B_4)
- ◇ Определение (j, B_1) **не достигает** блоков B_3 и B_4 , так как его убивает определение (j, B_2)

2.3.2.1 Достигающие определения: примеры

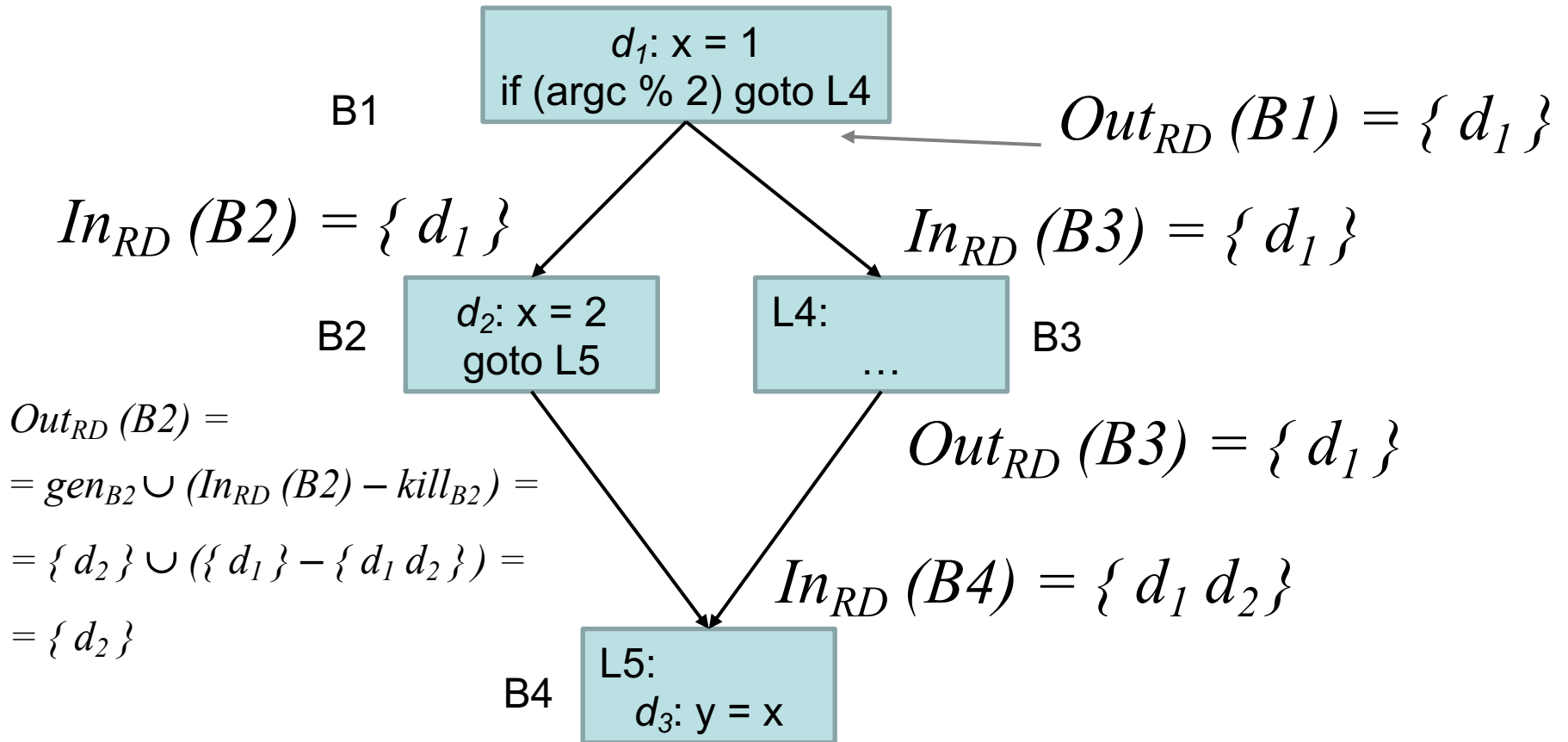


Достигают ли определения d_1 и d_2 блока B4?

2.3.2.1 Достигающие определения: примеры



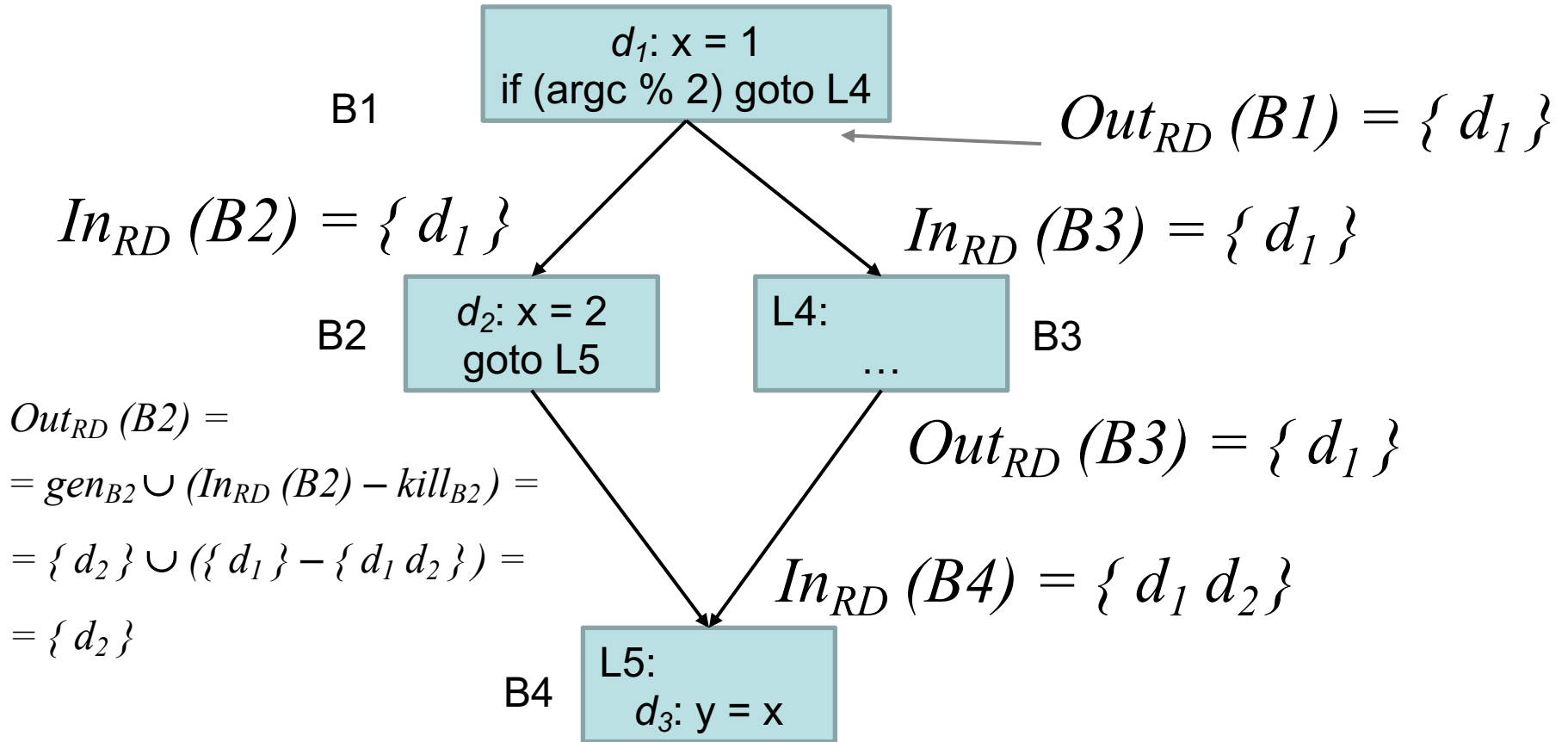
2.3.2.1 Достигающие определения: примеры



В общем виде: $Out_{RD}[B_i] = gen_{B_i} \cup (In_{RD}[B_i] - kill_{B_i})$

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} Out_{RD}[P]$$

2.3.2.1 Достигающие определения: примеры



Консервативность решения: если учесть "лишние" пути (и определения), это не приведет к некорректной оптимизации.

Например, если компилятор решает, можно ли распространить константу «2» в « d_3 »

2.3 Достигающие определения

2.3.3 Передаточные функции для достигающих определений

◇ Рассмотрим инструкцию I

$$d: \mathbf{u} = \mathbf{v} + \mathbf{w}$$

расположенную между точками p_1 и p_2 программы.

- ◇ Пусть x – множество определений, достигающих точки p_1
 gen_I – множество определений, порождаемых
инструкцией I
 $kill_I$ – множество определений, убиваемых инструкцией I
 y – множество определений, достигающих точки p_2
- ◇ $gen_I = \{d\}$
- ◇ для определения $kill_I$ нужно иметь **все другие определения \mathbf{u}** , т.е. несколько базовых блоков, а иногда и всю процедуру.

2.3 Достигающие определения

2.3.3 Передаточные функции для достигающих определений

- ◇ Рассмотрим инструкцию I

$$d: \mathbf{u} = \mathbf{v} + \mathbf{w}$$

расположенную между точками p_1 и p_2 программы.

- ◇ По определению передаточной функции

$$y = f_I(x)$$

- ◇ Инструкция I сначала убивает все предыдущие определения \mathbf{u} , а потом порождает d – новое определение \mathbf{u} .

Следовательно

$$y = gen_I \cup (x - kill_I)$$

- ◇ Следовательно, передаточная функция f_I инструкции I может быть записана в виде:

$$f_I(x) = gen_I \cup (x - kill_I)$$

2.3 Достигающие определения

2.3.4. Передаточные функции вида *gen-kill*

◇ **Определение.**

Передаточные функции, определяемые соотношением

$$f(x) = gen \cup (x - kill)$$

будем называть передаточными функциями вида *gen-kill*.

◇ **Утверждение 1.**

Композиция двух функций вида *gen-kill* является функцией вида *gen-kill*.

$$\begin{aligned}(f_2 \circ f_1)(x) &= f_2(f_1(x)) = \\ &= gen_2 \cup \left((gen_1 \cup (x - kill_1)) - kill_2 \right) = \\ &= gen_2 \cup (gen_1 - kill_2) \cup (x - kill_1 - kill_2) \\ (f_2 \circ f_1)(x) &= gen_{f_2 \circ f_1} \cup (x - kill_{f_2 \circ f_1})\end{aligned}$$

где $gen_{f_2 \circ f_1} = gen_2 \cup (gen_1 - kill_2)$

$$kill_{f_2 \circ f_1} = kill_1 \cup kill_2$$

2.3 Достигающие определения

2.3.4. Передаточные функции вида *gen-kill*

◇ **Утверждение 2.**

Пусть базовый блок B содержит n инструкций, каждая из которых имеет передаточную функцию $f_i(x) = gen_i \cup (x - kill_i)$
 $i = 1, 2, \dots, n$. Тогда передаточная функция для базового блока B может быть записана как

$$f_B(x) = gen_B \cup (x - kill_B) \quad ,$$

где

$$kill_B = kill_1 \cup kill_2 \cup \dots \cup kill_n$$

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \dots \\ \cup (gen_1 - kill_2 - kill_3 - \dots - kill_n)$$

2.3 Достигающие определения

2.3.5. Передаточные функции вида *gen-kill*

- ◇ Если какая-либо переменная определяется в блоке B несколько раз, то в gen_B войдет только ее последнее определение, т.е.

только последнее определение переменной будет действительно вне блока.

2.3 Достигающие определения

2.3.6. Система уравнений

- ◇ Таким образом, для КАЖДОГО базового блока B_i можно выписать уравнение

$$Out[B_i] = f_B(In[B_i])$$

или в случае анализа достигающих определений

$$Out[B_i] = gen_B \cup (In[B_i] - kill_B)$$

- ◇ Если ГПУ содержит n базовых блоков, получится n уравнений относительно $2 \cdot n$ неизвестных $In[B_i]$ и $Out[B_i]$, $i = 1, 2, \dots, n$.
- ◇ Еще n уравнений получится с помощью сбора вкладов путей.

2.3 Достигающие определения

2.3.6 Сбор вкладов путей

- ◇ Определение достигает точки программы, тогда и только тогда, когда существует по крайней мере один путь, вдоль которого эта точка может быть достигнута.

Этот путь должен пройти через какую-нибудь вершину из $Pred(B)$, причем если путь, проходящий через вершину $P \in Pred(B)$, не проходит ни через одну вершину, содержащую определение какой-либо переменной из B , то $Out(P) = \emptyset$. Следовательно

$$In[B] = \bigcup_{P \in Pred(B)} Out[P]$$

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

- ◇ Получается система уравнений

$$Out_{RD}[B_i] = gen_{B_i} \cup (In_{RD}[B_i] - kill_{B_i})$$

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} Out_{RD}[P]$$

($i = 1, 2, \dots, n$).

(RD - Reaching definitions)

- ◇ $In_{RD}[B]$ – множество переменных, определенных на входе в блок B
- ◇ $Out_{RD}[B]$ – множество переменных, определенных на выходе из блока B

В дальнейшем индекс RD будет опускаться

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

◇ Полученную систему уравнений

$$Out_{RD}[B_i] = gen_{B_i} \cup (In_{RD}[B_i] - kill_{B_i})$$

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} Out_{RD}[P]$$

можно упростить, произведя очевидную подстановку, в результате чего система уравнений примет вид:

$$In_{RD}[B_i] = \bigcup_{P \in Pred(B_i)} (gen_P \cup (In_{RD}[P] - kill_P))$$

$(i = 1, 2, \dots, n)$.

или, если вспомнить, что было обещано опускать индекс RD ,

$$In[B_i] = \bigcup_{P \in Pred(B_i)} (gen_P \cup (In[P] - kill_P))$$

$(i = 1, 2, \dots, n)$.

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

- ◇ Систему уравнений

$$In[B_i] = \bigcup_{P \in Pred(B_i)} (gen_P \cup (In[P] - kill_P))$$

$$(i = 1, 2, \dots, n)$$

будем решать методом итераций.

- ◇ При этом для одного или более блоков B_i множество $Pred(B_i)$ может содержать вершину $Entry$. В этих случаях будет использоваться «граничное условие» $In[Entry] = \emptyset$
- ◇ В качестве начальных итераций $In[B_i]$ возьмем пустые множества:
 $(In[B_i])^0 = \emptyset$

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

◇ Алгоритм «Достигающие определения»

- ◇ **Вход:** ГПУ (N, E) , в котором для каждого базового блока $B_i \in N$ вычислены множества $kill_{B_i}$ и gen_{B_i}
- ◇ **Выход:** множества $In[B_i]$, $(i = 1, 2, \dots, n)$ достигающих определений **НА ВХОДЕ** в каждый базовый блок B_i графа потока управления

- ◇ **Метод:** Используется **МЕТОД ИТЕРАЦИЙ** с начальной итерацией $(In[B_i])^0 = \emptyset$.

На всех итерациях r : $(In[Entry])^r = \emptyset$ (граничное условие)

Итерации продолжаются до тех пор, пока все множества $(In[B_i])^r$ (r – номер итерации) не перестанут изменяться.

- ◇ **Замечание.** Некоторые из множеств $(In[B_i])^r$ могут перестать изменяться гораздо раньше последней итерации.

2.3 Достигающие определения

2.3.7 Итеративный алгоритм для вычисления достигающих определений

```
In[Entry] =  $\emptyset$ ;  
change = true;  
for (каждый базовый блок B, отличный от Entry)  
    In[B] =  $\emptyset$ ;  
  
/* основной цикл*/  
while (change) do {  
    change = false;  
    for (каждый базовый блок B, отличный от Entry) {  
        /* вычисление новых значений In[B] и переменной  
        change */  
        
$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$
  
        if (InNew[B]  $\neq$  In[B]){  
            In[B] = InNew[B];  
            change = true;  
        }  
    }  
}
```

Алгоритм 2.3.7 не учитывает **замечания**: для некоторых значений i окончательный результат может быть получен намного раньше, чем будет выполнена последняя итерация. А это означает, что если на самом деле так получится, соответствующие значения будут много раз напрасно вычисляться, сравниваться с предыдущей итерацией и отвергаться – много лишней работы.

Получится, что алгоритм оптимизации сам не оптимален

```
while (change) do {
    change = false;
    for (каждый базовый блок  $B$ , отличный от  $Entry$ ) {
        /* вычисление новых значений  $In[B]$  и переменной
            $change$  */
         $InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$ 
        if ( $InNew[B] \neq In[B]$ ) {
             $In[B] = InNew[B]$ ;
            change = true;
        }
    }
}
```

Поэтому имеет смысл модифицировать алгоритм 2.3.7, введя понятие **рабочего множества**. Рабочее множество **WorkList**; представляет собой очередь, в которую помещаются только те базовые блоки, которые требуют дальнейшей обработки.

2.3 Достигающие определения

2.3.8 Модифицированный итеративный алгоритм для вычисления достигающих определений

In[Entry] = \emptyset ;

WorkList = \emptyset ;

for (каждый базовый блок В, отличный от Entry) {

 поместить В в WorkList;

 In[B] = \emptyset ; /* Каждому In[B] присваивается значение его нулевой итерации */

};

do { /* основной цикл*/

 Выбрать из очереди WorkList очередной блок В

 Вычислить InNew[B], используя уравнение

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

/*При вычислении InNew[B] множества In[P], где $P \in Pred(B)$, могут иметь значения либо текущей, либо следующей итерации

 if (InNew[B] \neq In[B]) {

 In[B] = InNew[B];

 Поместить Succs(B) в конец очереди WorkList

 }

} while |WorkList| > 0;

2.3 Достигающие определения

2.3.8 Модифицированный итеративный алгоритм для вычисления достигающих определений

```
In[Entry] = ∅;
```

```
WorkList = ∅;
```

```
for (каждый базовый блок В, отличный от Entry) {
```

```
    поместить В в WorkList;
```

```
    In[B] = ∅; /* Каждому In[B] присваивается значение его  
                нулевой итерации */
```

Ценность алгоритма в том, что каждый базовый блок рассматривается не на каждой итерации, а столько раз, сколько он попадает в WorkList

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

```
/*При вычислении InNew[B] множества In[P],  
где P ∈ Pred(B), могут иметь значения либо  
текущей, либо следующей итерации
```

```
if (InNew[B] ≠ In[B]) {
```

```
    In[B] = InNew[B];
```

```
    Поместить Succs(B) в конец очереди WorkList
```

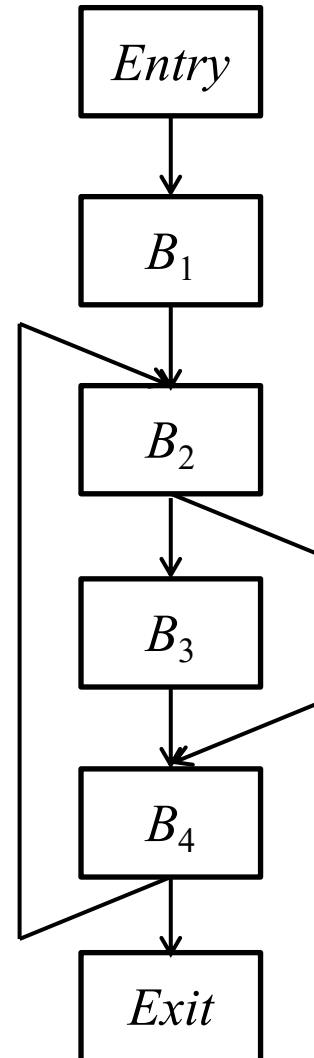
```
}
```

```
} while |WorkList|>0;
```

2.3 Достигающие определения

2.3.8 Пример

B_1 $i \leftarrow -, m, 1$ $j \leftarrow n$ $a \leftarrow u1$	B_2 $i \leftarrow +, i, 1$ $j \leftarrow -, j, 1$
B_3 $a \leftarrow u2$	B_4 $i \leftarrow u3$



2.3 Достигающие определения

2.3.8 Пример

В простой программе справа 7 определений. Требуется определить, какие определения достигают входов в ее 5 базовых блоков **B1**, **B2**, **B3**, **B4** и **Exit**

B_1 $d_1: i \leftarrow -, m, 1$ $d_2: j \leftarrow n$ $d_3: a \leftarrow u1$	B_2 $d_4: i \leftarrow +, i, 1$ $d_5: j \leftarrow -, j, 1$
B_3 $d_6: a \leftarrow u2$	B_4 $d_7: i \leftarrow u3$

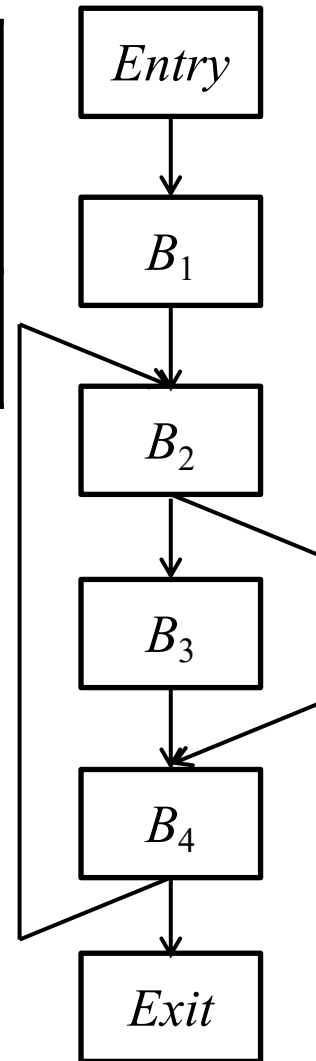
$d_1 = (i, B_1)$ – определение переменной i в блоке B_1

$d_4 = (i, B_2)$ – определение переменной i в блоке B_2

$d_7 = (i, B_4)$ – определение переменной i в блоке B_4

Вычислим множества gen и $kill$ для каждого базового блока

Для блока B_1 $gen_{B_1} = \{d_1, d_2, d_3\}$ $kill_{B_1} = \{d_4, d_5, d_6, d_7\}$



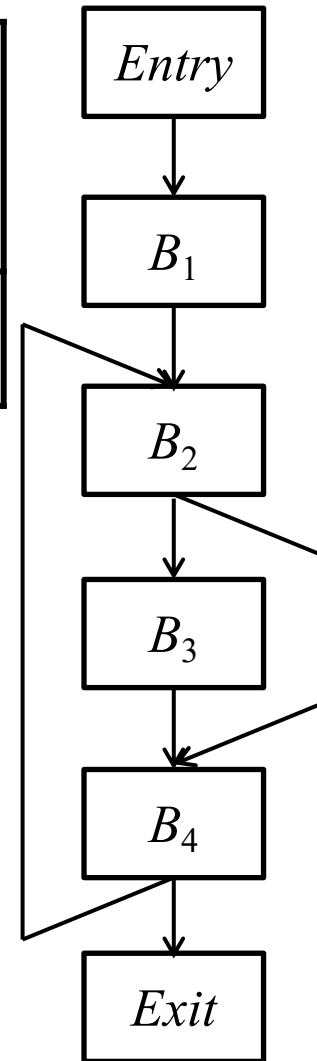
2.3 Достигающие определения

2.3.8 Пример

B_1 $d_1: i \leftarrow -, m, 1$ $d_2: j \leftarrow n$ $d_3: a \leftarrow u1$	B_2 $d_4: i \leftarrow +, i, 1$ $d_5: j \leftarrow -, j, 1$
B_3 $d_6: a \leftarrow u2$	B_4 $d_7: i \leftarrow u3$

Результаты вычисления множеств *gen* и *kill* для базовых блоков сведены в таблицу

B	gen_B	$kill_B$
B_1	$\{(i, B_1), (j, B_1), (a, B_1)\}$ = (1110000)	$\{(i, B_2), (j, B_2), (a, B_3), (i, B_4)\}$ = (0001111)
B_2	$\{(i, B_2), (j, B_2)\}$ = (0001100)	$\{(i, B_1), (j, B_1), (i, B_4)\}$ = (1100001)
B_3	$\{(a, B_3)\} = (0000010)$	$\{(a, B_1)\} = (0010000)$
B_4	$\{(i, B_4)\} = (0000001)$	$\{(i, B_1), (i, B_2)\} = (1001000)$
<i>Exit</i>	\emptyset	\emptyset



2.3 Достигающие определения

2.3.8 Пример

B_1 $d_1: i \leftarrow -, m, 1$ $d_2: j \leftarrow n$ $d_3: a \leftarrow u1$	B_2 $d_4: i \leftarrow +, i, 1$ $d_5: j \leftarrow -, j, 1$
B_3 $d_6: a \leftarrow u2$	B_4 $d_7: i \leftarrow u3$

Множества удобно представлять битовыми векторами, длина которых равна мощности базового множества. В рассматриваемом примере длина векторов равна 7.

$$gen_{B_1} = (1110000) \quad kill_{B_1} = (0001111)$$

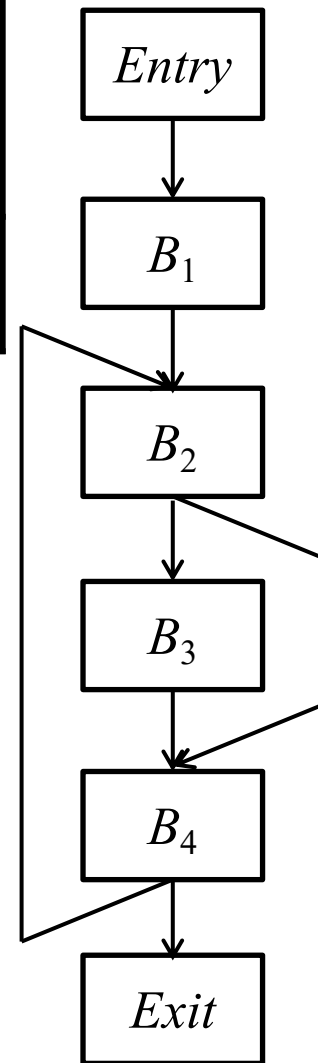
d_1 d_2 d_3 d_4 d_5 d_6 d_7

(i, B_1) (j, B_1) (a, B_1) (i, B_2) (j, B_2) (a, B_3) (i, B_4)

Определения примера 2.3.8

B	gen_B	$kill_B$
B_1	(1110000)	(0001111)
B_2	(0001100)	(1100001)
B_3	(0000010)	(0010000)
B_4	(0000001)	(1001000)

Множества gen и $kill$ базовых блоков примера 2.3.8



2.3 Достигающие определения

2.3.8 Пример

Нулевая итерация:

$$(In[B_i])^0 = \emptyset = (00000000), \quad i = 1, 2, 3, 4.$$

$$(In[Exit])^0 = \emptyset = (00000000)$$

Первая итерация:

Вычисляем $(In[B_i])^1$ ($i = 1, 2, 3, 4$) и $(In[Exit])^1$ по формуле

$$InNew[B] = \bigcup_{P \in Pred(B)} (gen_P \cup (In[P] - kill_P))$$

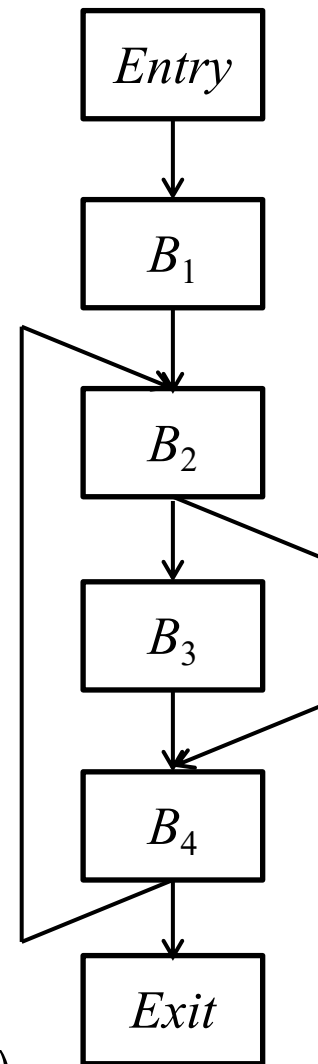
используя значения $(In[B_i])$, известные к моменту вычисления

Вычисление $(In[B_1])^1$. $(In[B_1])^1 = \emptyset$ $(In[B_1])$ не попадает в WorkList и больше не будет вычисляться

Вычисление $(In[B_2])^1$.

$$\begin{aligned} (In[B_2])^1 &= (gen_{B_1} \cup ((In[B_1])^1 - kill_{B_1})) \cup (gen_{B_4} \cup ((In[B_4])^0 - kill_{B_4})) = \\ &= ((1110000) \cup (\emptyset - (0001111))) \cup ((0000001) \cup (\emptyset - (1001000))) = \\ &= (1110000) \cup (0000001) = (1110001) \end{aligned}$$

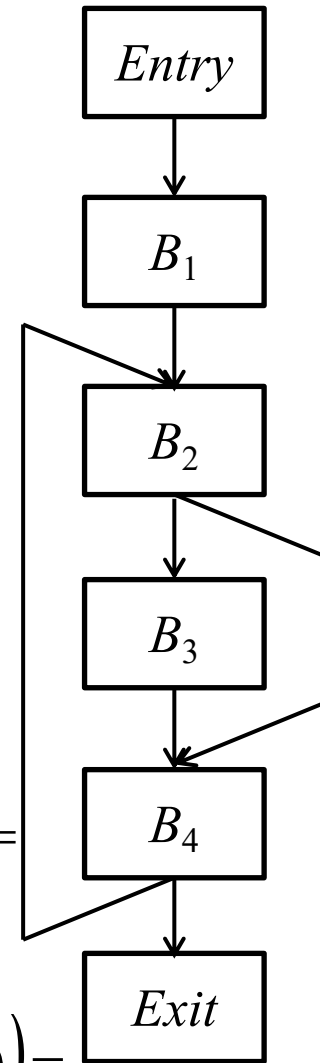
B	$Pred(B)$
B_1	$Entry$
B_2	$\{B_1, B_4\}$
B_3	B_2
B_4	$\{B_2, B_3\}$
$Exit$	B_4



2.3 Достигающие определения

2.3.8 Пример

B	gen_B	$kill_B$	$Pred(B)$
B_1	(1110000)	(0001111)	$Entry$
B_2	(0001100)	(1100001)	$\{B_1, B_4\}$
B_3	(0000010)	(0010000)	B_2
B_4	(0000001)	(1001000)	$\{B_2, B_3\}$



Первая итерация:

Вычисление $(In[B_3])^1$. $(In[B_3])^1 = gen_{B_2} \cup ((In[B_2])^1 - kill_{B_2}) =$
 $= (0001100) \cup ((1110001) - (1100001)) =$
 $= (0011100)$

Вычисление $(In[B_4])^1$.

$$(In[B_4])^1 = (gen_{B_2} \cup ((In[B_2])^1 - kill_{B_2})) \cup (gen_{B_3} \cup ((In[B_3])^1 - kill_{B_3})) =$$

$$= ((0001100) \cup ((1110001) - (1100001))) \cup$$

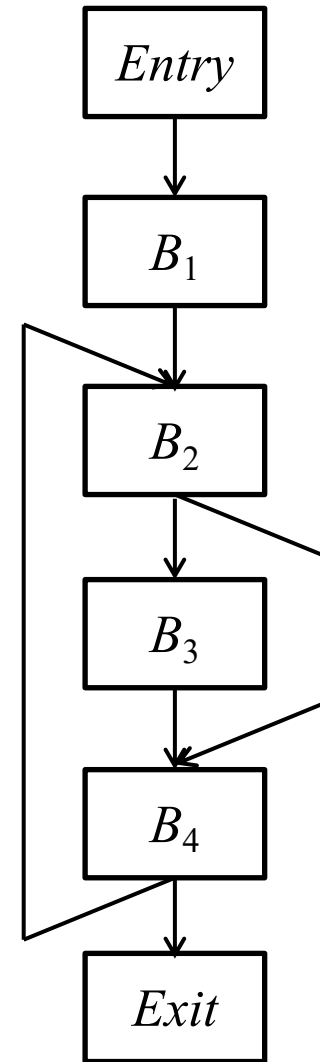
$$\cup ((0000010) \cup ((0011100) - (0010000))) =$$

$$= (0011110)$$

2.3 Достигающие определения

2.3.8 Пример

B	gen_B	$kill_B$	$Pred(B)$
B_1	(1110000)	(0001111)	$Entry$
B_2	(0001100)	(1100001)	$\{B_1, B_4\}$
B_3	(0000010)	(0010000)	B_2
B_4	(0000001)	(1001000)	$\{B_2, B_3\}$



Первая итерация:

Вычисление $(In[Exit])^1$.

$$\begin{aligned}(In[Exit])^1 &= gen_{B_4} \cup ((In[B_4])^1 - kill_{B_4}) = \\ &= (0000001) \cup ((0011110) - (1001000)) = \\ &= (0010111)\end{aligned}$$

2.3 Достигающие определения

2.3.8 Пример

B	$(In[B])^1$	$(In[B])^2$	$(In[B])^3$
B_1	0000000	0000000	0000000
B_2	1110001	1110111	1110111
B_3	0011100	0011110	0011110
B_4	0011110	0011110	0011110
$Exit$	0010111	0010111	0010111

Вторая и третья итерации выполняются аналогично (их результаты внесены в таблицу). На этом процесс завершается, так как, как видно из таблицы, все $(In[B])^3$ совпадают с $(In[B])^2$, т.е. *WorkList* пуст.

2.3.9 Множества *Input* для базовых блоков

- ◇ Множество $Input[B]$ для базового блока B – это множество $In_{RD}[B]$, которое строится при исследовании достигающих определений

2.4 Живые переменные

2.4.1 Множества *Output* для базовых блоков

- ◇ Множества *Output* для базовых блоков строятся как результат анализа, позволяющего выявить *живые переменные*, т.е. переменные, используемые в базовых блоках, в которые управление попадает после выхода из исследуемого базового блока.
- ◇ Анализ похож на предыдущий, но ГПУ просматривается не с начала, а с конца: от *Exit* к *Entry*.

2.4.2. Определение

- ◇ Цель анализа – для определения переменной x в точке p программы выяснить, будет ли указанное значение x использоваться вдоль какого-нибудь пути, начинающегося в точке p .
 - ◇ Если да – переменная x *жива* (активна) в точке p ,
 - ◇ если нет – переменная x *мертва* (неактивна) в точке p .

2.4 Живые переменные

2.4.3 Уравнения потока данных

◇ $In_{LV}[B]$ – множество переменных, **живых** на входе в блок B

LV – Live Variables

$Out_{LV}[B]$ – множество переменных, **живых** на выходе из блока B .

◇ В чем проблема?

Пусть в блоке B используется переменная v .

Возможны 2 случая:

1) используется определение v в одном из блоков $B' \in Pred^*(B)$;

2) используется определение v в самом блоке B .

В первом случае говорят, что v жива на выходе из B'

Во втором случае говорят, что v мертва на выходе из B'

2.4 Живые переменные

2.4.3 Уравнения потока данных

◇ $In_{LV}[B]$ – множество переменных, **живых** на входе в блок B

LV – Live Variables

$Out_{LV}[B]$ – множество переменных, **живых** на выходе из блока B .

◇ def_B – множество переменных, определяемых в блоке B до их использования в этом блоке

(любая переменная из def_B **мертва** на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred_B$)

◇ use_B – множество переменных, используемых в блоке B до их определения в этом блоке

(любая переменная из use_B **жива** на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred_B$)

◇ **Замечание.** В анализе живых переменных рассматриваются не определения переменных, а сами переменные.

2.4 Живые переменные

2.4.3 Уравнения потока данных

◇ $In_{LV}[B]$ – множество переменных, **живых** на входе в блок B

LV – Live Variables

$Out_{LV}[B]$ – множество переменных, **живых** на выходе из блока B .

◇ def_B – множество переменных, определяемых в блоке B до их использования в этом блоке

(любая переменная из def_B **мертва** на входе в блок B и, следовательно на выходе каждого блока $B' \in Pred_B$)

В качестве def_B можно рассматривать множество всех переменных, определяемых в блоке (а не только определяемых до использования) – на результат вычисления $Out_{LV}[B]$ это не повлияет, т.к. они также входят в use_B и в передаточной функции будут добавлены независимо от того, есть ли они в def_B :

$$In[B] = use_B \cup (Out[B] - def_B)$$

$$Out[B] = \bigcup_{S \in Succ(B)} In[S]$$

2.4 Живые переменные

2.4.3 Уравнения потока данных

Рассмотрим блок B

$m \leftarrow 1$
$k \leftarrow m + k$
$a \leftarrow a + b + m$

Речь идет о выходе из блоков из $Pred(B)$

Для этих блоков $m \in def$

Остальные переменные $\in use$

Если переменная встречается и в правой, и в левой части присваивания, то для определений def и use можно считать, что правая часть выполняется раньше левой.

2.4 Живые переменные

2.4.3 Уравнения потока данных

B_1 $i \leftarrow -, m, 1$ $j \leftarrow n$ $a \leftarrow u1$	B_2 $i \leftarrow +, i, 1$ $j \leftarrow -, j, 1$
B_3 $a \leftarrow u2$	B_4 $i \leftarrow u3$

◇ Пример

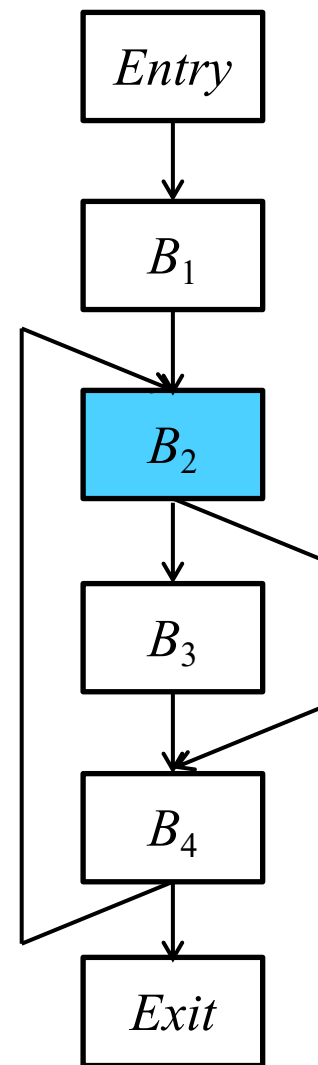
(1) в блоке B_2 переменные i и j используются до их переопределения, следовательно,

$$use_{B_2} = \{i, j\} = (\mathbf{11000000})$$

(2) в блоке B_2 определяются новые значения переменных i и j , так что

$$def_{B_2} = \{i, j\} = (\mathbf{11000000})$$

В программе 8 переменных: $i, j, a, m, n, u1, u2, u3$



2.4 Живые переменные

2.4.3 Уравнения потока данных

- ◇ Уравнения, связывающие *def* и *use* с неизвестными *In* и *Out*, определяются следующим образом:

$$In[B] = use_B \cup (Out[B] - def_B)$$

$$Out[B] = \bigcup_{S \in Succ(B)} In[S]$$

- ◇ К ним добавляется граничное условие

$$Out[Exit] = \emptyset$$

2.4 Живые переменные

2.4.3 Уравнения потока данных

- ◇ Уравнения, связывающие def и use с неизвестными In и Out , определяются следующим образом:

$$In[B] = use_B \cup (Out[B] - def_B)$$

$$Out[B] = \bigcup_{S \in Succ(B)} In[S]$$

def_B – множество переменных, определяемых в блоке B до их использования* в этом блоке (**Исключаем заведомо мертвые переменные**).

use_B – множество переменных, используемых в блоке B до их определения** в этом блоке (**Добавляем новые живые переменные**)

(*) Также можно рассматривать в качестве def_B просто множество всех переменных, определяемых в блоке B – на результат это не повлияет.

(**) В случае use_B важно, что переменные используются до определения в блоке.

Хотя по своему смыслу определения действительно должны быть симметричны, допустимость (*) объясняется только видом передаточной функции.

2.4 Живые переменные

2.4.3 Уравнения потока данных

- ◇ Если значение In , определяемое первым уравнением подставить во второе уравнение, множества In будут исключены из системы уравнений и получится система уравнений, содержащая в качестве неизвестных только множества Out :

$$Out[B] = \bigcup_{S \in Succ(B)} (use_S \cup (Out[S] - def_S))$$

- ◇ К ним добавляется граничное условие

$$Out[Exit] = \emptyset$$

2.4 Живые переменные

2.4.4 Итеративный алгоритм анализа живых переменных

◇ Алгоритм «Живые переменные»

- ◇ **Вход:** ГПУ, в котором для каждого блока B вычислены множества def и use
- ◇ **Выход:** множества переменных, живых на выходе ($Out[B]$) каждого базового блока B .
- ◇ **Метод:** выполнить следующую программу:

2.4 Живые переменные

2.4.4 Итеративный алгоритм анализа живых переменных

◇ *Алгоритм «Живые переменные»*

◇ **Метод:** выполнить следующую программу:

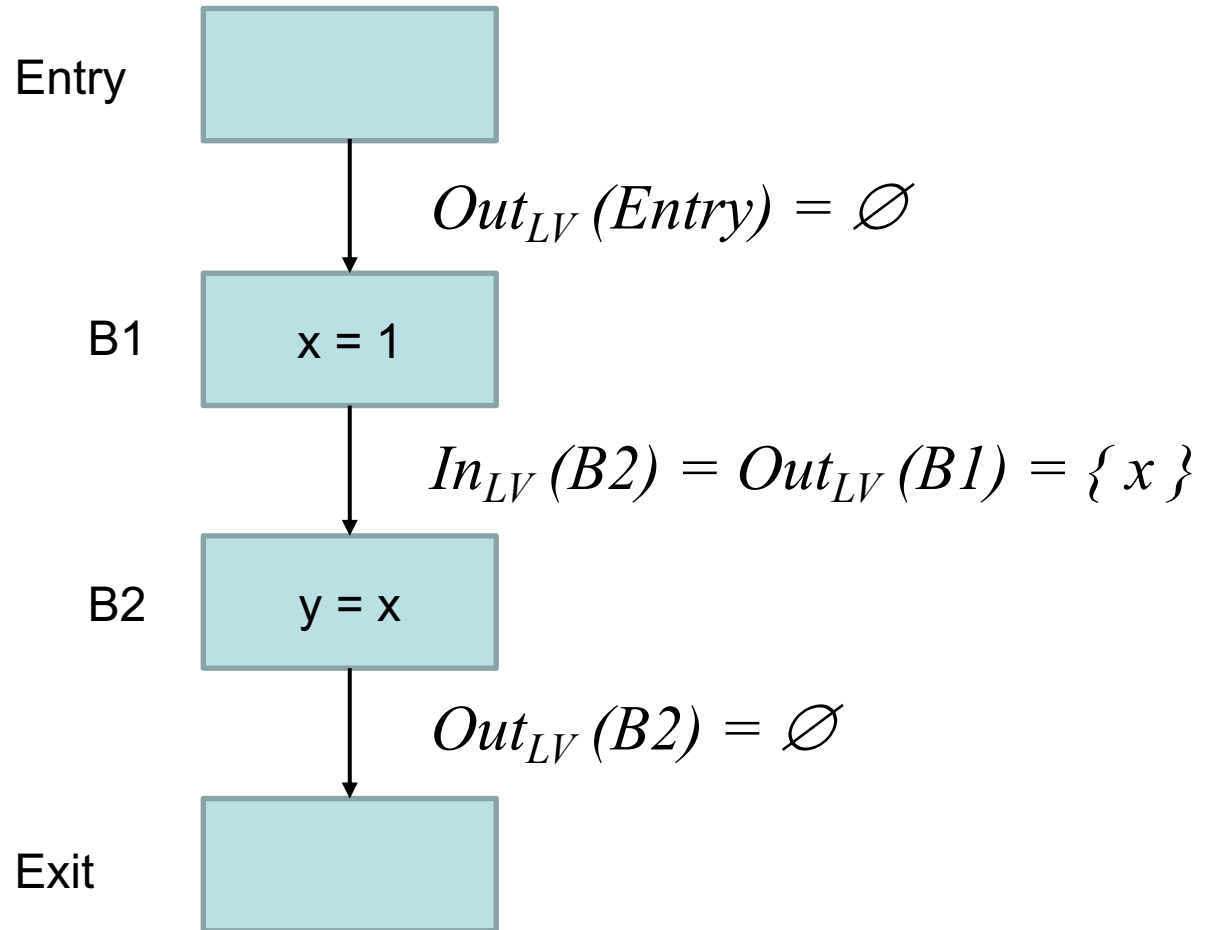
```
Out[Exit] = ∅;
WorkList = ∅;
for(каждый базовый блок B, отличный от Exit) {
    Out[B] = ∅;
    WorkList ∪= {B}
}
/*основной цикл*/
do { Извлечь блок B из WorkList (исключив B);
    OutNew[B] =  $\bigcup_{S \in \text{Succ}(B)} (\text{use}_S \cup (\text{Out}[S] - \text{def}_S))$ 
    if (OutNew[B] ≠ Out[B]) {
        Out[B] = OutNew[B];
        WorkList ∪= Pred(B)
    }
} while |WorkList| > 0;
```

2.4 Живые переменные

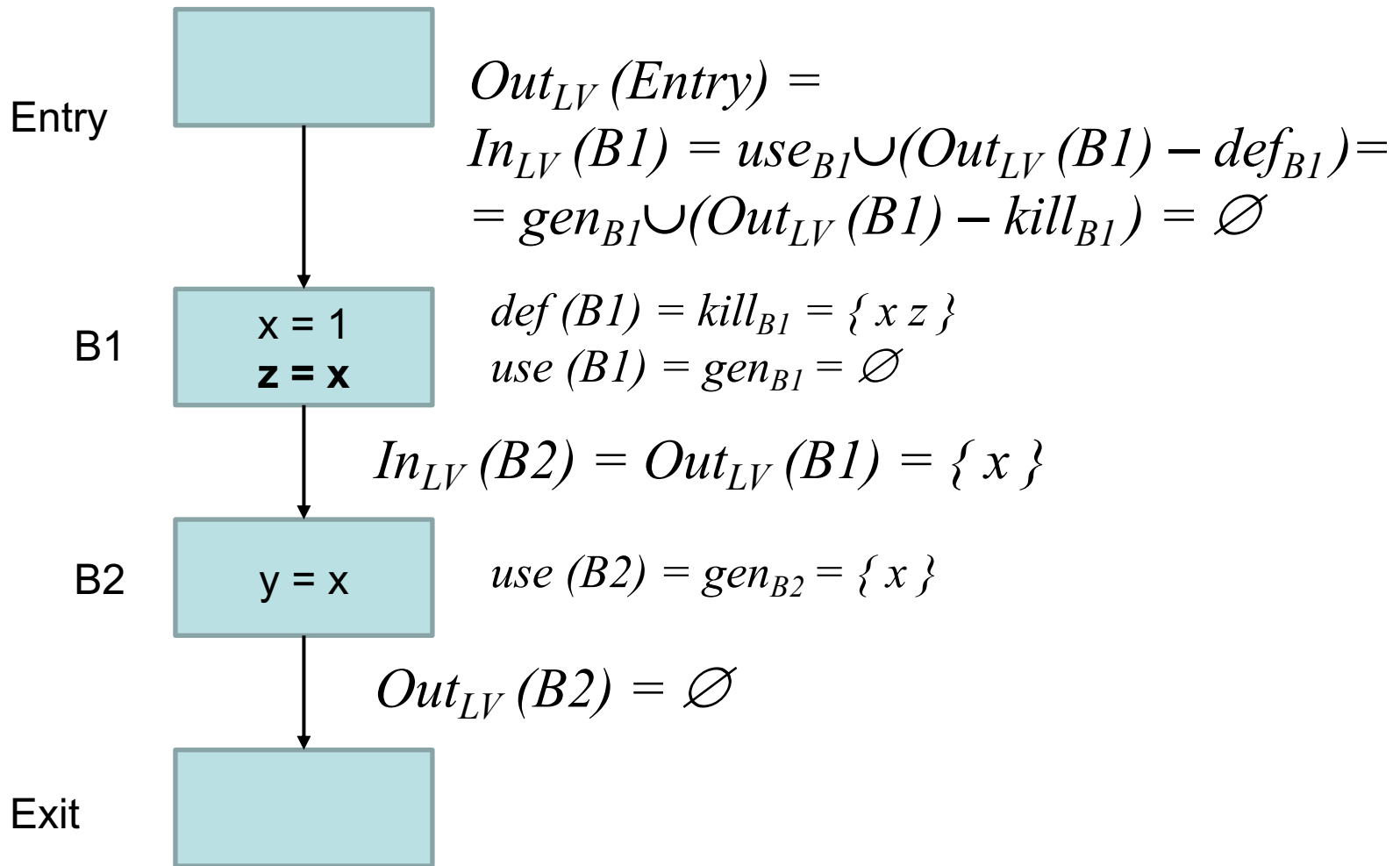
2.4.5 Множества *Output* для базовых блоков

- ◇ Множество $Output[B]$ для базового блока B – это множество $Out_{LV}[B]$, которое строится при исследовании живых переменных

2.4.6 Примеры: живые переменные

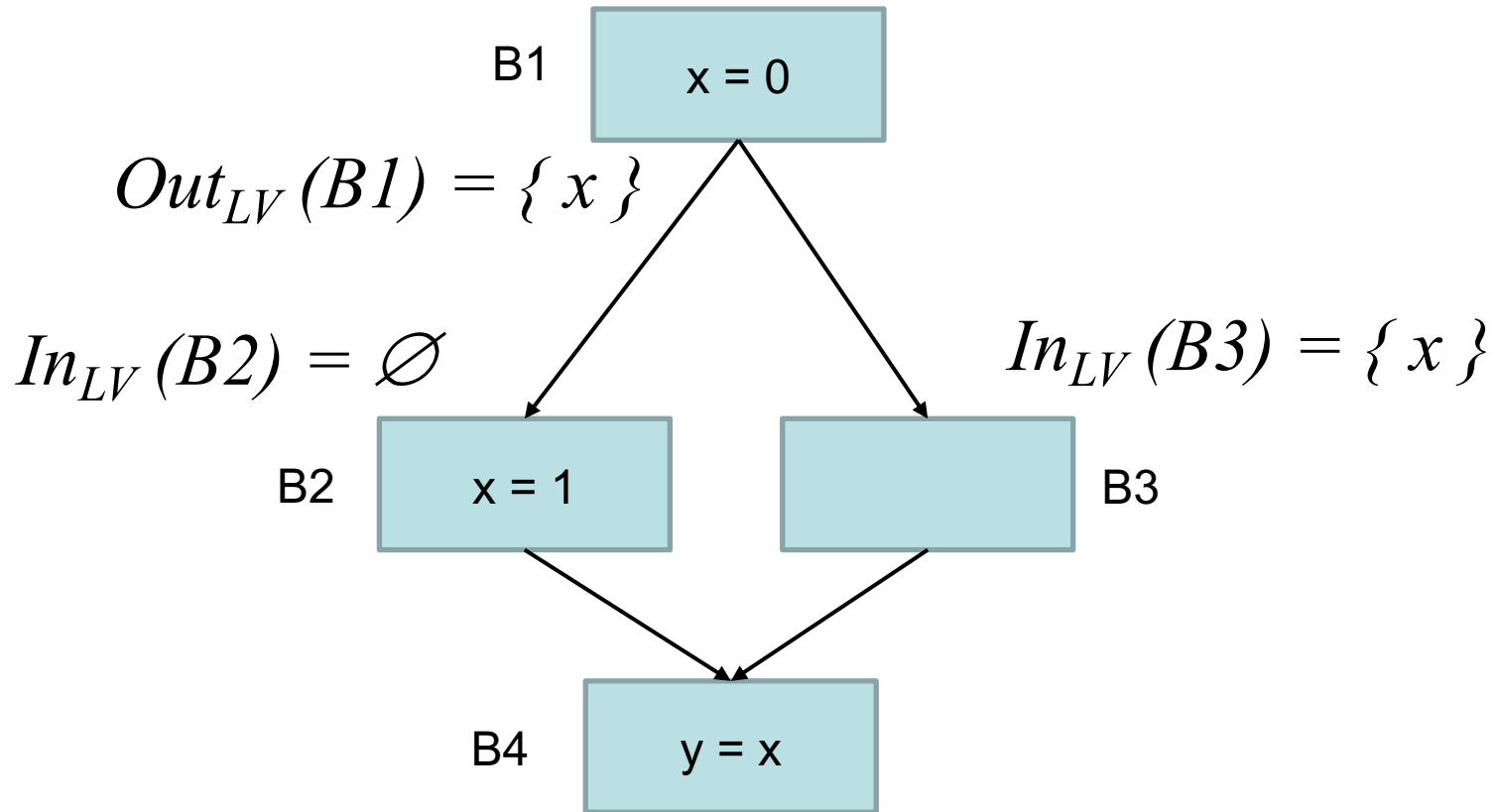


2.4.6 Примеры: живые переменные



Множества *def* и *kill*, а также *use* и *gen* для данной задачи являются синонимами. В общем виде для данного класса задач (*Gen-Kill*) анализа потоков данных используется *gen/kill*, в частном случае для живых переменных – *use/def*.

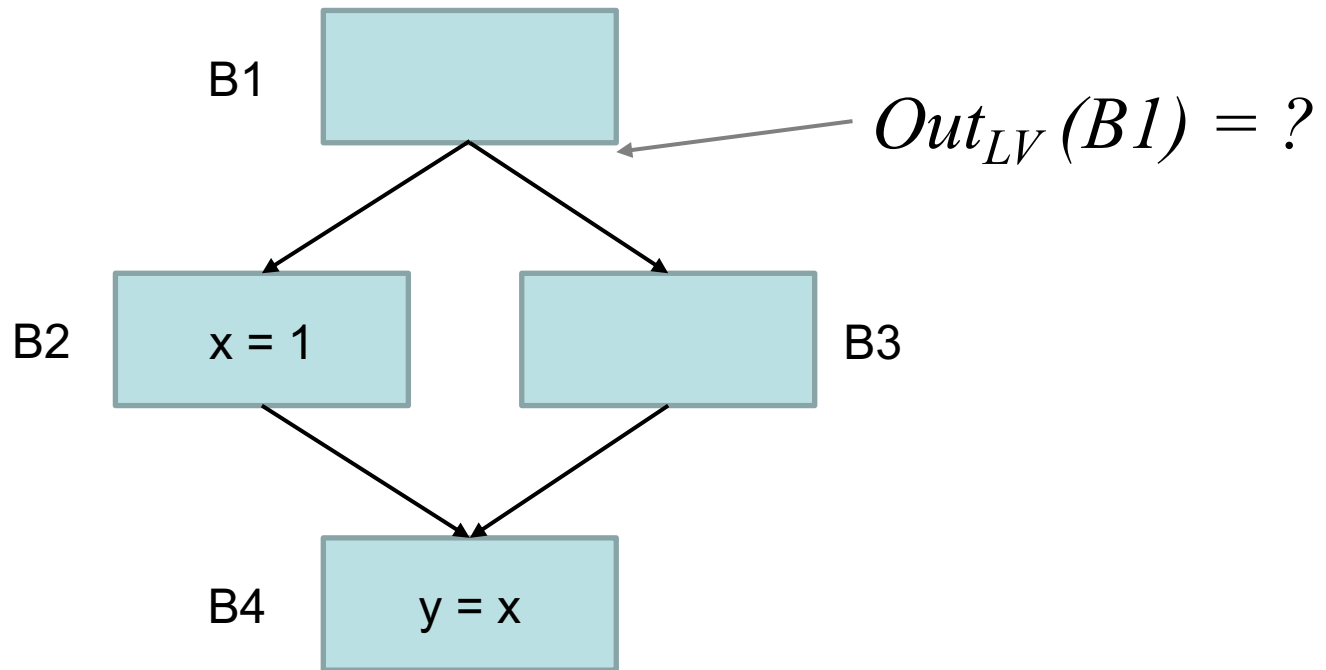
2.4.6 Примеры: живые переменные



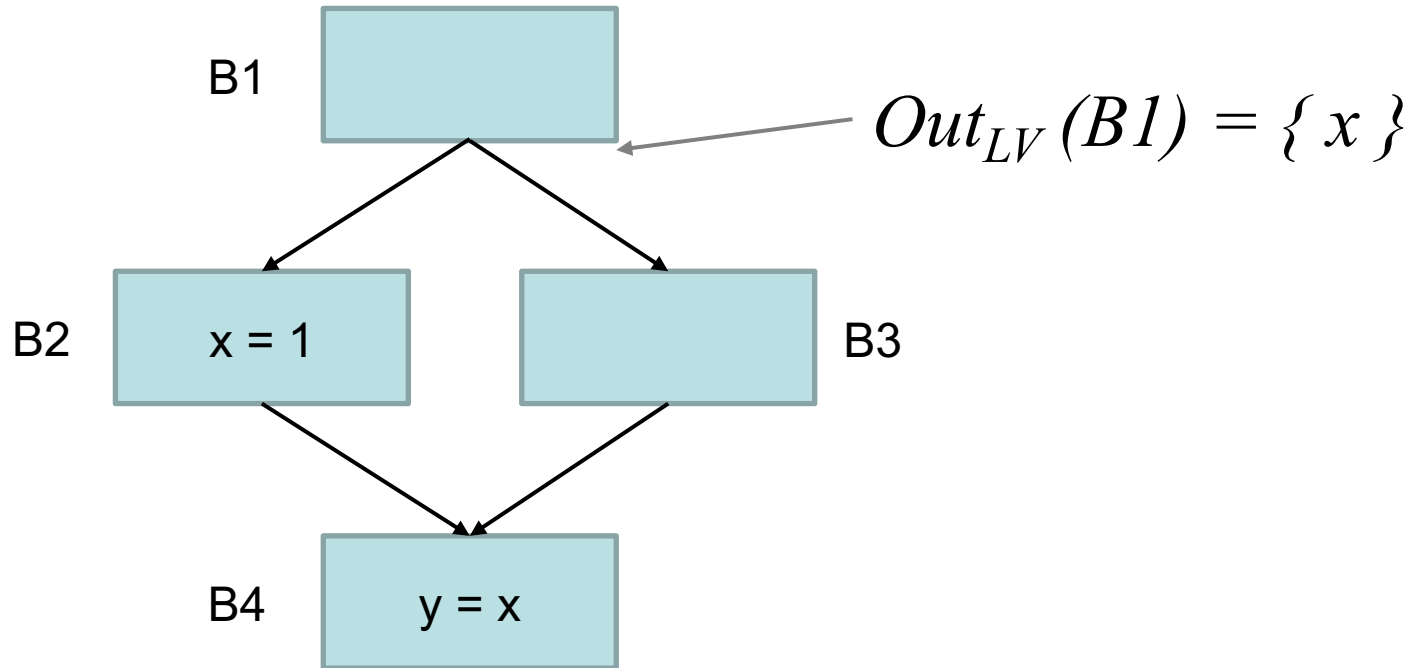
Консервативность решения: если учесть "лишние" пути (и определения), это не приведет к некорректной оптимизации.

Например, если компилятор решает, можно ли в B3 занять регистр, в котором лежит «x», под другую переменную.

2.4.6 Примеры: живые переменные



2.4.6 Примеры: живые переменные



Строго говоря, живая переменная может вообще не иметь определения в рассматриваемом ГПУ.